Collecting and visualizing business metrics in cloud-based applications

Juhana Suhonen

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 30.6.2017

Thesis supervisor:

Assoc. Prof. Keijo Heljanko

Thesis advisor:

M.Sc. (Tech.) Heikki Moilanen

Aalto University
School of Science

AALTO UNIVERSITY
SCHOOL OF SCIENCE

ABSTRACT OF THE
MASTER'S THESIS

Author: Juhana Suhonen

Title: Collecting and visualizing business metrics in cloud-based applications

Date: 30.6.2017 Language: English Number of pages: 7+65

Computer Science and Engineering

Professorship: Computer Science

Supervisor: Assoc. Prof. Keijo Heljanko

Advisor: M.Sc. (Tech.) Heikki Moilanen

Monitoring cloud computing resources is a straightforward and common task for any cloud application developer. The problem with current monitoring solutions is that they only focus on infrastructure resources. Many companies on the other hand would need data about the business side of their applications. This thesis extends the current monitoring solutions to capture business metrics from within applications. The metrics are then visualized to quickly allow for better analysis of the data.

The tool is composed of three main components. The metrics are captured with a Node.js library that is imported in the monitored application. The library sends the captured data to InfluxDB timeseries database. The data is visualized with Grafana which implements tables, graphs, and gauges. The provided command-line tool creates a file that can be imported in Grafana to create a new dashboard with graphs in it.

The requirements for the tool were created through the needs of software developers and clients of web- and mobile-developer Codemate. An architectural design was made based on the requirements and then implemented on the AWS cloud platform on top of Kubernetes. The implementation was evaluated by testing it in a real production server.

The tool is functional and it works as intended. The results from the evaluation prove that the tool created in this thesis can help companies gain better information about their products. Future work includes adding the metrics capture for other languages such as Go and Ruby as well as integrating the tool to Codemate's new development environment. Further research can be done especially in improving performance of the solution in large systems.

Keywords: business metrics, monitoring, distributed system, grafana, kubernetes

AALTO-YLIOPISTO
PERUSTIETEIDEN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Juhana Suhonen

Työn nimi: Bisnesmetriikan kerääminen ja visualisointi pilvipohjaisessa kehitysympäristössä

Päivämäärä: 30.6.2017 Kieli: Englanti Sivumäärä: 7+65

Tietotekniikka

Professuuri: Tietotekniikka

Työn valvoja: Prof. Keijo Heljanko

Työn ohjaaja: DI Heikki Moilanen

Pilviresurssien monitorointi on selkeä ja yleinen tehtävä jokaiselle pilvipalvelun kehittäjälle. Monitorointisovellukset keskittyvät vain infrastruktuuriresursseihin, vaikka monet nykyajan yritykset tarvitsisivat tarkempaa tietoa sovellusten bisnespuolesta. Tämä diplomityö laajentaa nykyisiä monitorointisovelluksia kattamaan bisnesmetriikan keräämisen applikaatioiden sisältä sekä visualisoi datan paremman analyysin mahdollistamiseksi.

Diplomityössä kehitetty työkalu koostuu kolmesta osasta. Metriikat kerätään sovelluksista Node.js-kirjaston avulla, joka lisätään sovelluksen koodiin. Kirjasto lähettää dataa InfluxDB-tietokantaan, josta se visualisoidaan Grafanalla interaktiivisten kuvaajien sekä taulukoiden avulla. Grafanaan voidaan lisäksi luoda työpöytiä diplomityötä varten luodulla ohjelmalla.

Bisnesmetriikan keräämiseen ja visualisointiin luotu työkalu määriteltiin ohjelmistokehittäjä Codematen ohjelmistoinsinöörien sekä asiakkaiden tarpeiden mukaan. Määrittelyä käytettiin työkalun arkkitehtuurin luomiseen, joka ohjasi käytännön toteutusta. Työkalu rakennettiin Amazonin AWS-palveluun Kuberneteksen päälle. Toteutetun työkalun toimivuus testattiin lopuksi asiakasympäristössä tuotantopalvelimella.

Työkalun todettiin toimivan tarkoituksenmukaisesti. Testauksesta saadut tulokset osoittavat, että työkalu voi auttaa yrityksiä saamaan parempaa informaatiota ohjelmistotuotteistaan sekä niiden käytöstä. Työkalun kehitystä voidaan jatkaa laajentamalla sen toimintaa Go- ja Ruby-kielille sekä integroimalla se tiiviimmin Codematen uuteen kehitysympäristöön. Lisätutkimus erityisesti suorituskyvyn parantamiseen laajoissa järjestelmissä on tarpeen.

Avainsanat: bisnesmetriikka, monitorointi, hajautettu järjestelmä, grafana, kubernetes

Preface

Writing this thesis has been a flowing process ever since I got first hints at the topic from Codemate in late January. With my interest going towards full-stack development, it has been rewarding to work on a project that offers aspects of both front- and back-end. Working on a multi-disciplinary topic has also been a great way to get to know several architects and software engineers at Codemate.

The thesis has progressed steadily even with other projects requiring most of my attention at times. A big thanks for this goes to the Codemate team, the relaxed work environment and Rocket League providing valuable relaxation during writing sprints. On the writing part, my instructor Heikki has helped improve my writing and this thesis tremendously. Without his guidance this thesis would not be as fluent as it is now. My professor Keijo Heljanko has also been very supportive even with the limited time he has been able to provide to my thesis.

For the implementation part, Tuomas Mäkinen has helped me a lot with the issues I have faced, and he has also given great guidance. To allow me to test the implementation in real production and for the always friendly help and attitude, I want to thank Petteri and Juho. Finally, for all the mental, physical and textual support, I want to thank Bembu, Henry, Petri, Joel, Laura and the many others who have helped me with the thesis and my studies.

This thesis has been written with the Dvorak key-layout. Trying to write the thesis with Dvorak at 15 words per minute was slow and straining at first, but I got faster every week. Learning it in daily 10-minute doses has also been fun and challenging, keeping my mind fresh when writing. Now, when finishing this thesis in the end of June, my writing speed has increased to 40 wpm. It is still slower than I am with QWERTY (50wpm), but it is much easier to code with Dvorak. Starting to learn Dvorak slowly has been a great decision, and I encourage others to do so as well. Over time, this will no doubt increase the typing speed and ergonomics tremendously compared to the normal layout.

Otaniemi, June 30th, 2017

Juhana Suhonen

Contents

Abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
AWS	Amazon Web Services
BA	Business Analytics
BI	Business Intelligence
BPEL	Business Processing Expression Language
BPM	Business Process Management
EBS	Amazan Elastic Block Store
EC2	Amazon Elastic Compute Cloud
ELB	Amazon Elastic Load Balancer
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
IoT	Internet of Things
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
MF	Measurement Framework
MFW	Monitoring Framework
MVP	Minimum Viable Product
NIST	U.S. National Institute of Standards and Technology
PaaS	Platform as a Service
PPI	Process Performance Indicator
REST	Representational State Transfer
SaaS	Software as a Service
SBA	Service-Based Architecture
SC	Service Component
SDE	Software Development Environment
SI	Service Infrastructure
SLA	Service Level Agreement
SOA	Service-Oriented Architecture
SQL	Structured Query Language
SRS	Software Requirements Specification
VM	Virtual Machine

1 Introduction

1.1 Motivation

Cloud computing has become very popular in recent years. It provides scalable computing resources over the Internet to perform large-scale and complex computing. [45, 31] The main advantages of cloud computing compared to traditional approaches include virtualized resources, parallel computing and reduced costs [31].

The cloud computing paradigm can be divided into three main categories as shown in Figure 1: Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS) [41]. SaaS services are applications hosted by vendors and include for example Google Drive and Dropbox [33]. PaaS offers more customization, allowing the users to develop products of their own on top of the platform. Examples of PaaS include Google App Engine and Microsoft Azure. [24] IaaS refers to a combination of hosting, hardware provisioning and basic services needed to run a cloud, including services like Amazon and IBM Smart Cloud [37, 24].

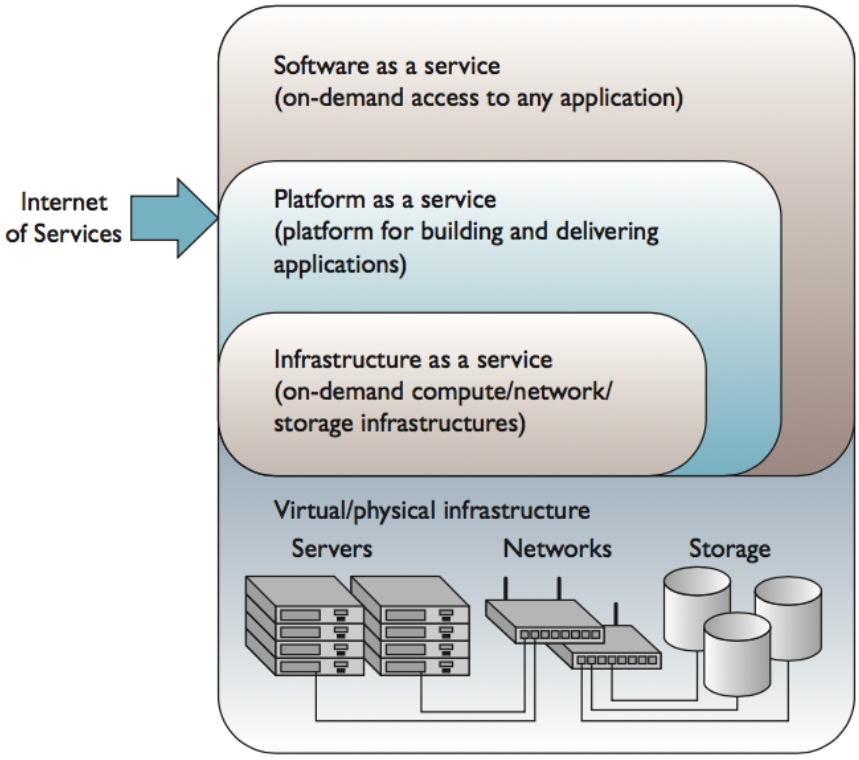

Figure 1: Cloud Computing Model [41]

Manvi and Shyam [37] identify IaaS as one of the most significant and fastest growing fields in cloud computing. The IaaS platforms allow for efficient usage of hardware resources, such as processing and storage units, while minimizing the costs by distributing those computing resources to several users [24]. IaaS also allows the developer to control the whole stack of their application from infrastructure to

software. It can simplify development significantly thanks to the distributed nature of IaaS which allows the developers to develop for a single platform. [37].

There are also several problems with IaaS, such as inefficient resource management [37, 54] and the lack of universal tools that support IaaS [11]. According to Komi-Sirviö and Tihinen [35], projects are most often undermined by poor software development environments and tools. In their study, tool-related issues were the most commonly experienced problem area, with it being an issue in 81% of the studied cases.

Some more specific tools have been created to help in distributed development [11, 29, 32]. Most of them target the development process, management or communication. Tools that help with the development activity itself are few, mostly including some programming environments and online repositories [29].

Monitoring the resource usage of cloud applications has been the focus of many researchers [8, 13, 18, 25]. However, little effort has been put into allowing the users of the cloud to choose what to monitor: "there is a clear lack of monitoring software which can be easily installed and integrated in a cloud computing infrastructure" [18, p. 68]. Furthermore, most current monitoring solutions have been designed for simple systems or focus on monitoring the physical hardware rather than application or business metrics chosen by the user.

1.2 Research problem and thesis scope

To tackle the problems with creating individual development environments, web and mobile developer Codemate has started to build an environment of their own. It will include several tools for creating and managing the cloud resources for development projects. The cloud production environment is meant to simplify development and keep track of all the different projects underway. The aim of Codemate's environment is to allow the creation, management and monitoring of all the different projects that are being developed in the company.

The new development environment should allow both infrastructure and application metrics to be monitored. Currently, monitoring infrastructure resources is a common and straightforward task that every developer running applications in the cloud must do. It is focused completely on the devops (development and IT operations) and does not provide any real value to the application, business or end-users. In order to operate a business effectively however, data about the application itself and its users are needed as well. This field has gained much less attention.

This thesis aims to design and develop a cloud-based environment for gathering business metrics from applications running in the cloud. The system should capture, store and visualize metrics chosen by the developers easily and effectively, and scale to applications of different sizes. The environment should give the developers simple tools for capturing the business metrics directly from their applications. It should capture metrics such as the amount of sales of an online store or the number of users logging in to the system at any given time.

The result of this thesis will be part of the new cloud production environment for Codemate. The main focus will be in describing how the metrics can be gathered,

stored and visualized, and then implementing a tool to do that. Dhingra et al. [26] state that existing frameworks have not looked at cloud monitoring from the point of view of cloud customers where monitoring is based on customer needs. This thesis addresses this lack of research through the following research questions:

1. What kinds of metrics should the environment support?
2. How can the current resource monitoring systems be extended to monitor business metrics?
3. What are the requirements and design of the tool based on the needs of the developers and end-users?
4. Does the tool provide a good way to gather and analyze business metrics?

The thesis will start with a literature review on current distributed development and monitoring solutions in Chapter 2. It will also discuss business process measurements and how to visualize them. Chapter 3 presents the requirements and Chapter 4 the design and architecture for the cloud-based metrics environment. Chapter 5 explains how the tool was created and how it works. Chapter 6 provides a reflection in order to evaluate if the implemented methods were effective. It also presents the results of testing the tool in a real production environment. Finally, the work will be summarized and future improvements discussed in Chapter 7.

2 Literature review

2.1 Definitions of cloud computing

To understand the basis for this thesis, a closer look must be taken into various different fields that come together in cloud application monitoring and gathering business metrics. Beginning with the definitions for cloud computing and business metrics, we move on to distributed development, monitoring and the current solutions on the market and in research.

Vaquero et al. [30, p. 51] have defined cloud computing as follows: "Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized Service Level Agreements (SLAs)." The definition is in line with the U.S. Department of Commerce's National Institute of Standards and Technology's (NIST) definition for cloud computing [39].

Figure 2 shows the general cloud computing stack. At the bottom lies the network infrastructure through which individual pieces of computing infrastructure communicate with each other. Atop the infrastructure is the platform which works as the middle-ware between the infrastructure and individual applications. The platform can contain virtualized resources or operating systems, among other things. The applications support and enable the business processes that are run in the cloud.

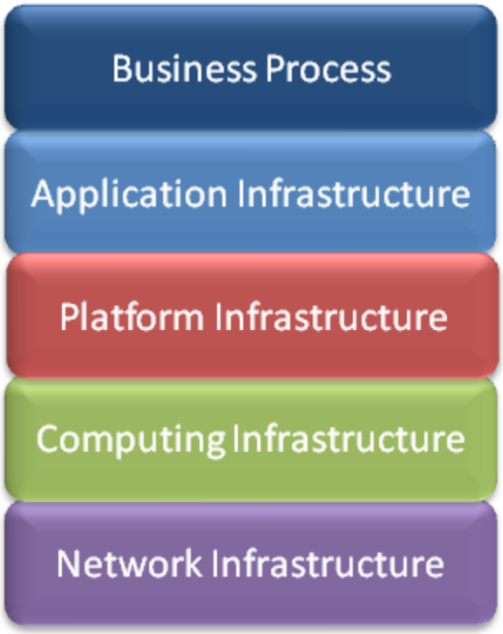

Figure 2: Cloud Computing Stack [41]

Cloud computing has emerged as a big field in the last ten years since Google introduced their cloud computing platform [55]. Da Silva et al. [24] explain that the concept of cloud computing has gained attention recently mainly because of the industry's initiative: Large IT infrastructure companies had a considerable amount of their hardware idle waiting for consumption peaks. Thus, they attempted to optimize and even profit from their computing infrastructure, which lead to the creation of related concepts and proprietary technology for cloud computing. The first academic papers dealing explicitly with cloud computing started to appear around 2008-2009 [24, 30].

There are several architectures and guiding principles within cloud computing. Two main ones are Service-Based Architectures (SBA) and Service-Oriented Architectures (SOA). SBAs are composed of a number of loosely coupled services available on the network which provide the desired functionalities. SOA on the other hand utilizes independent component services with standard interfaces as the basic construct. In essence, SBA is composed of smaller individual services than SOA, allowing for a more granular control over individual pieces of the application. [46]

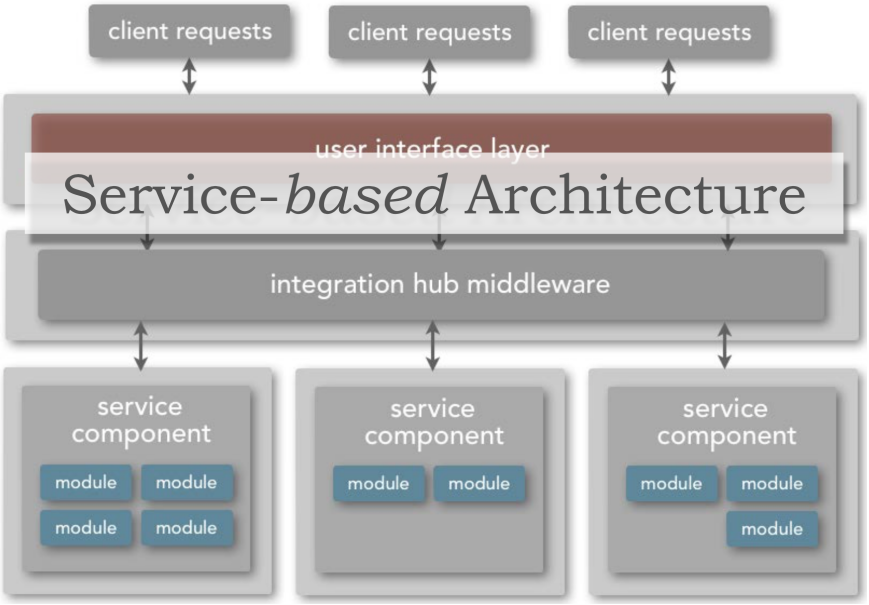

Figure 3: Service-based architecture [27]

Figures 3 and 4 show examples of SBA and SOA as described by Neal Ford [27]. In both SBA and SOA, services are autonomous and platform independent computational entities that can be used in heterogeneous environments. A SBA can be viewed by its three functional layers as Business Process Management (BPM) Layer, Service Composition (SC) Layer and Service Infrastructure (SI) Layer. [46] While in the past monitoring has mostly concentrated on the Service Infrastructure layer, this thesis aims to broaden monitoring all the way to the BPM layer.

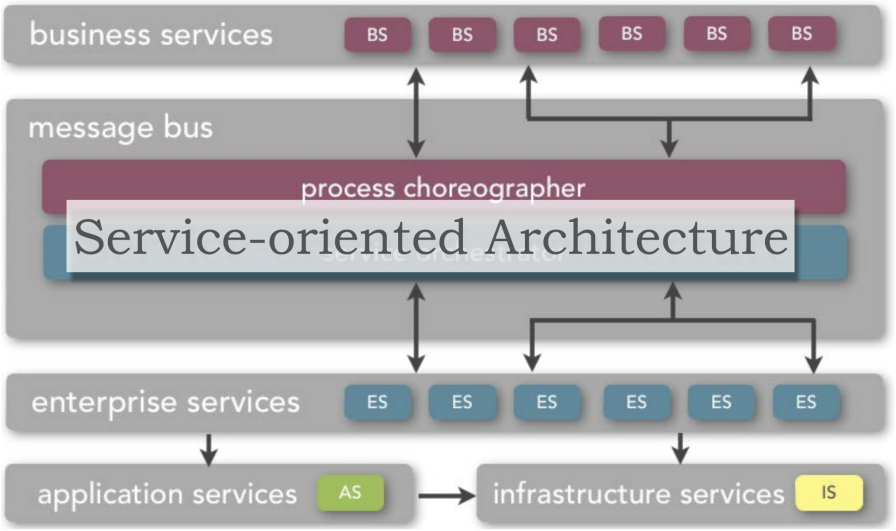

Figure 4: Service oriented architecture [27]

2.2 Business metrics

Business metrics are meant to give data on which to act in the decision-making process for a business. Business metrics are quantifiable measurements that track and assess the status of a specific business process. Every area of a business has specific performance metrics that should be monitored. [5]

According to software engineering theories [43, p. 434], "a measure provides a quantitative indication of the extent, amount, dimensions, capacity, or size of some attribute of a product or a process", and "measurement is the act of determining the measures". Furthermore, an attribute states what we want to measure while the metrics define how we measure something [40].

A *base measure* is the measure obtained executing a certain measurement method over a single process instance, such as counting the number of visitors on a website. An *aggregated measure* uses a set of process instances to calculate values for the Key Performance Indicator (KPI). The aggregation is usually performed by selecting all the instances in a given interval of time, but also by selecting all the instances for a specific attribute and afterwards applying a certain aggregation function like max, min, average or sum on this set of measures to obtain one single value. [21]

While business metrics are used to give an overall view of the business, KPIs can be used to provide more detailed analysis of the critical areas for performance. Further, Process Performance Indicators (PPI) are quantifiable metrics that can evaluate the efficiency and effectiveness of business processes. PPIs can be measured directly from the generated data within the process flow and are aimed at the process controlling and continuous optimization. [21]

Within the scope of this thesis, we are specifically interested in data gained from the applications run in the cloud. The data should help businesses better track their performance, so it must adhere to the formats required by KPIs and PPIs. When

developing these applications, the developers must not only think about the business side of the application, but also the technical side. With relation to business metrics, data capturing and distributed development are important topics.

2.3 Distributed development

The continuous increase in the volume and detail of data has produced an overwhelming flow of data in both structured and unstructured format. The data is typically captured or created by organizations like Google and Facebook, and consists of content such as that created in the social media, by devices part of the Internet of Things (IoT), and multimedia. Data creation is occurring at a record rate. A major challenge for researchers and practitioners is that this growth rate exceeds their ability to design appropriate cloud computing platforms for data analysis and update-intensive workloads. [31] The increased amount of data also requires the developers and managers to communicate more with each other and the parties using the data.

Difficulties in communication and collaboration can come from the nature of distributed development which often spans over multiple teams and sites [22]. To mitigate such problems, there are different types of software development environments (SDE) available [16]. They come in different types, ranging from SDEs for individual developers to multi-site SDEs that can support the whole project [35]. In the past, software development tools and environments only focused on the task of coding [11]. Since then, more focus has been put into collaboration and communication.

A major part of a development project is the development environment. Every cloud development project traditionally needs an isolated environment in the cloud in order to develop and test their service. Setting up these individual production environments for each project can be problematic. Their configuration takes time, and there is no simple way of keeping track of the utilization of resources or cost of all the projects combined. [41] Ideally, the developers would have a unified development environment through which setting up individual projects is easy.

Creating a uniform software development environment is a challenging task. The readiness to change development tools is also low, indicating that people want to use the tools they are familiar with. [35] This might be one reason, why only few large-scale cloud providers have managed to establish themselves on the market. These providers offer monitoring tools of their own, blocking third-party tools from their infrastructure. The tools are not very good however, mostly focusing on the needs of the service provider instead of their clients.

2.4 Monitoring in cloud applications

Monitoring is a technique for software information collection, behavior diagnosis, defect detection and status recovery [13]. Since the early 1960s, software monitoring has been successfully used in many areas including debugging and testing, correctness checking, performance evaluation and enhancement, and security and dependability analysis and control [13, 56].

Monitoring can take place at various stages in the software life cycle. One common monitoring discipline is runtime software monitoring. It is "the act of collecting some information about a system during its operation" [34, p. 155]. It has been used for profiling, performance analysis, software optimization and fault-detection, diagnosis and recovery [25, 56]. Monitoring is most often related to the infrastructure resources, but also applications themselves can be monitored.

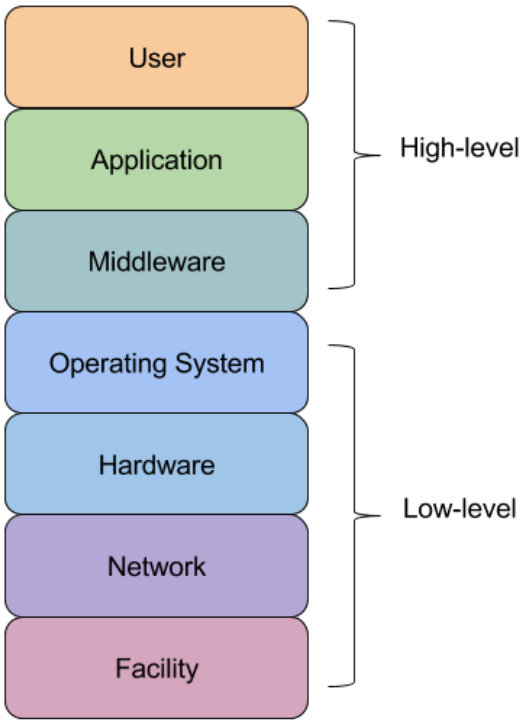

Figure 5: Monitoring in the 7 layers of cloud computing

According to the Cloud Security Alliance [10, 52, 53], cloud can be divided into seven layers as shown in Figure 5: Facility, Network, Hardware, Operating System, Middleware, Application and User. Monitoring in these layers can be categorized as high- and low-level monitoring [7]. Low-level monitoring concentrates on the hardware and operating system, containing the first four layers. High-level monitoring collects information at the middleware, application and user layers.

In the context of cloud monitoring, these layers can be seen as where monitoring can be done. The first three layers deal with hardware while the last two provide data from within the individual applications. High-level monitoring information is generally of more interest for the end-user. [7]

Cloud computing resource monitoring is beneficial to the end-users because it helps them analyze their resource requirements. It also ensures that they get the requested amount of resources they are paying for. Cloud resource monitoring enables the end-users to know when to request more resources, when to release any underutilized resources, and how much resources to reserve for any specific task. [26]

Many monitoring tools in use currently have not been designed for cloud usage. As such, traditional monitoring solutions have several flaws. First, they do not fit in the life-cycle of virtual resources. Second, they are usually based on monitoring agents installed inside the physical machines to gather metrics. Finally, many current monitoring solutions are only suitable for small-scale deployments due to their poor scalability. [18]

The lack of information and control is one of the main challenges associated with cloud computing monitoring. Especially the customization of the monitoring metrics that the cloud customers have over the rented cloud resources needs improving. [18] At the time of writing, none of the existing frameworks have looked thoroughly at Cloud Monitoring from the point of view of cloud customers, where monitoring is

based on customer needs [26]. Due to lack of universal tools, it is especially difficult to monitor application data, and there are no clear guidelines for it, forcing every company to develop methods of their own.

2.5 Measuring business processes

To get the best results for the tool implemented in this thesis, it is important to understand how business processes can and should be measured. According to Seufert [48, p. 31], the main purpose of an analytics system is "to provide insight on product performance to product teams". Effective software measurement and meaningful data interpretation depend on recognizing the essential duality of all measurement processes. Measurement involves the definition of two models: the empirical, real-world context in which the measurement is to take place and a numerical model incorporating well-defined measurement-based aspects of the empirical model. [42]

The overall measurement process is shown in Figure 6. Once the goals of the measurement have been understood and defined, an empirical model can be created. Measuring according to the model leads to the numerical model which gives the numerical results after the necessary statistical functions have been applied. Interpreting the numerical results leads to the empirical results, which can be used to further refine the model. [42]

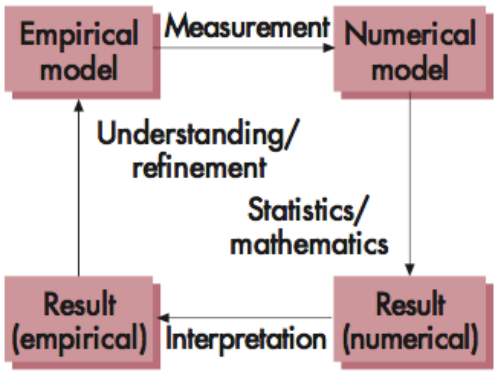

Figure 6: Measurement process [42]

In Service Oriented Architecture (SOA) approaches, business processes are usually accomplished by service-based applications [21]. Because of that, processes are represented in terms of technical service composition and their measurement is realized through KPI- monitoring of web-service compositions. Cardoso [21] identifies several different metrics that can be measured:

- *Time measure* measures the duration of time between two time instants.
- *Linear measure* takes into account the first occurrence of a given time instant condition and the last occurrence of another time instant condition.

- *Cyclic measure* takes into account pairs of time instant conditions that are located in a loop. They compute the KPI- measure by aggregating the time values of all the iterations.
- *Count measure* is a simple counter measuring the number of times something happens.
- *Condition measure* measures the fulfillment of certain condition in running or finished process instance.
- *Data measure* measures the value of a certain part of a data object.

Business Analytics (BA) brings all the different measurements together. Shmuelli et al. [49, p. 3] identify Business Analytics as "the practice and art of bringing quantitative data to bear on decision making." For example, Washington Post can track its readers by time of day, location, and user subscription information. Various methods exist for data analysis in BA. [49]

To make use of business analytics, it is important to have the right data available. Currently in many businesses, data is not being gathered or analyzed in an efficient manner. According to Mendonca and Basili [40], it is not uncommon to find software organizations that are either collecting redundant data, collecting data that nobody uses or collecting data that might be useful to people who do not even know it exists inside their organization.

Cardoso [21] mentions that there is a considerable gap on modeling performance indicators and integrating them into modern enterprise modeling frameworks. Therefore, the automated support for KPI- measurement is seriously impaired as well as the measurement of goal achievement. He also points out that the problem of capturing KPI- related metrics in accordance with strategic concerns is an issue commonly neglected in the current languages and frameworks. [21]

Business analytics have evolved to Business Intelligence (BI) in order to understand what has happened and is happening. Business Intelligence puts emphasis on understanding the captured data. BI uses charts, tables and dashboards to visualize, examine and explore data. Generic and static reports that were used before have been replaced with more user-friendly and effective tools and practices such as interactive dashboards. [49]

2.6 Visualizing metrics

Visualization is the graphical representation of data or concepts and it is the most valuable sense for providing information from computers to humans. This is mostly because humans acquire more information through vision than through all the other senses combined. [47] The human brain is much better able to locate and isolate selected events and activities when presented in graphical form in both real-time and post-mortem analysis modes [17].

Card et al. [20] identify six major methods in which visualization can amplify cognition by perception:

- Increasing memory and processing resources by allowing storage of massive amounts of information in a quickly accessible form.
- Reducing searching by grouping information together.
- Enhancing recognition of patterns by enhancing patterns.
- Perceptual inference by making some problems obvious.
- Perceptual monitoring by allowing monitoring of a large number of potential events.
- Manipulable medium by allowing exploration of a space of parameters unlike static diagrams.

The main purpose of data visualization is to view analytical results presented visually through different graphs for decision making [31]. For example, according to Seufert [48], the ability for a product team to group users by specific behaviors or demographic characteristics, or to limit a view of data by date, is an absolute necessity. One of the best ways to visualize and display data is by using dashboards which not only display the data but also allow the users to interact with it.

Shmuelli et al. [49, p. 4] explain that "effective dashboards are those that tie directly into company data, and give managers a tool to quickly see what might not readily be apparent in a large complex database". They identify one example as a tool for industrial operations managers that displays customer orders by showing customer name, type of product, size of order, and length of time to produce [49].

Several tools for visualizing metrics have been created. Savola and Heinonen [47] identify two tools that deal with software metrics: Prefuse and Streamsight. Prefuse is a toolkit for interactive information visualization, and Streamsight is a visualization tool for large-scale streaming applications. The former is however already outdated and the latter works only with streams.

The tools identified by Offen and Jeffrey [42] fail to address key aspects of software measurement, especially those concerning empirical models and reuse. According to them, automated context capture, data collection, and analysis, along with historical data, can greatly enhance the effectiveness of predictive estimation models. Furthermore, the organizations lacked support for data storage and analysis in all their case studies [42].

To address the problem of collecting redundant data, Mendonca and Basili [40] propose the solution to focus on two key issues: to better understand and structure the ongoing measurement and to better explore the data that the organization has already collected. A key aspect in this is the usage of a database that is designed for the specific task.

2.7 Databases

Databases are used to efficiently store and organize data. There are several different types of databases built for different use cases. Two main categories of databases are

the Structured Query Language (SQL) and NoSQL databases. SQL- databases are traditional databases which use relations and require strict rules for how and what type of data can be stored. NoSQL- databases on the other hand are much more relaxed in their requirements for the structure of data, and can even accept different types of data in the same tables. NoSQL- databases were designed for large Internet services such as Facebook and Twitter, and can handle massive amounts of data and requests. [23, p. 10-11]

The databases are often categorized by the activity they are used for. An *operational database*, or a transactional database, mostly supports the day-to-day operations of businesses. An *analytical database* focuses on storing historical data and business metrics that are used for tactical or strategic decision making. [23, p. 9] Analytical databases have become more important recently, being the core technological element in the field of business intelligence [23, Chapter 13].

Databases can be separated by the type of data they store. Most databases operate on key-value basis where each key in the database table has a distinct, single value. Some databases can store more arbitrary data such as documents, time-series data or even objects. This thesis is interested in the time-series databases which organize their data based on the timestamp attached to the stored value.

Databases have very different performances depending on the data they store, how it is organized in the database, and what they have been developed to do. For example, one database can be very fast at reading data but very slow at writing it, or vice versa. Zareian et al. [57] suggest that the data storage for a measurement framework should handle fast ingestion rate, low latency reads and rich data model. For the measurement work that they implemented, they selected a NoSQL- solution because they are "generally highly scalable, can relax ACID (Atomicity, Consistency, Isolation, Durability) in favor of performance and support a notion of free schema" [57, p. 60].

As the amount of data increases, the database may no longer be able to effectively store it all in real time. In such a situation, a queuing service such as BigQueue proposed by Zareian et al. [57] may come to question. Their experiments show that the BigQueue can improve database write speeds by up to 300 times. Such a solution might provide important performance enhancements for large systems also within the scope of this thesis, depending on the monitoring solution that is implemented.

2.8 Current monitoring solutions

Several different types of monitors have been created over the years. This chapter will first explain several ways to create monitors. It will then display different monitoring solutions that have been developed, showcasing the problems each has had.

According to Delgado et al. [25], there are several approaches for implementing monitors. The key differences in them come from monitoring points, placement, platform and implementation. *Monitoring points* explain at which point of code execution monitoring is initiated; manually in the code, or automatically by tools that detect points of instrumentation. *Placement* refers to where the monitoring code actually executes; inline between code execution, or offline in another thread or even on another machine. [25]

Platform differentiates between software and hardware; a software monitor uses code to observe and analyze the values of monitored variables whereas a hardware monitor might be a microprocessor that is attached to the physical system. *Implementation* details how the monitor is executed; in a single process, in a separate thread, or on a completely different processor. [25]

Delgado et al. [25] identify three different source program types where the monitoring tool is applied to. *General purpose* means that the monitoring system does not know anything about the target program. *Domain-specific* type includes some information about the target system, such as where it is used. *Category-specific* type indicates that the monitor is targeting a specific kind of program, such as one made for real-time or distributed systems. [25]

Monitoring typically disturbs the observed system to some extent, which is commonly referred to as the probe effect [34]. The dynamic behavior of software or hardware is recorded by a collection of sensors. They are placed in the user's program or run as separate processes. Each sensor is a section of code which sends information concerning an event or state within the program to the monitor. [50]

To collect the metrics, different types of probes can be identified and used for different systems. For example, Wang et al. [56] define four types of probes:

- Instrumented probes with analysis: probes embedded in a system that process raw data before outputting it.
- Instrumented probes without analysis: probes embedded in a system that provide the raw data as captured.
- Intercepting probes with analysis: external probes that process raw data before outputting it.
- Intercepting probes without analysis: external probes that provide the raw data as captured.

Instrumentation is the most widely used monitoring mechanism, where the monitoring code is embedded inside the target code [56]. Traditionally, the instrumentation code is inserted manually by the programmers, which allows the code to be inserted freely into any location of the monitored code. Another reason for using instrumentation is that it does not need support from the platform. The disadvantage of embedding monitoring code inside application code is that the probe code may make the service more difficult to maintain. [56]

There are several open-source and commercial products available for monitoring. Many of the products only concentrate on monitoring the infrastructure in very specific systems such as Amazon Web Services (AWS). Some popular products currently in use are VMDriver which gathers information by using the hypervisor, Nagios which was created for traditional IT- infrastructures, and Cloudwatch and OpenNebula Monitoring System which were designed for cloud computing, but are very limited [18]. Ganglia [38] is a scalable distributed monitoring system for high performance computing systems such as clusters and grids. It is highly efficient in large systems, but only concentrates on monitoring infrastructure resources.

Monasca [28] is an open-source, multi-tenant, highly scalable, performant and fault-tolerant monitoring-as-a-service solution that integrates with OpenStack. It uses a representational state transfer (REST) application programming interface (API) for high-speed metrics processing and querying and has a streaming alarm and notification engines. Monasca concentrates on monitoring infrastructure metrics. It is based on a pluggable design, where different components such as the database or visualization tool can easily be swapped. [28]

Bai et al. [13] have proposed an agent-based architecture for monitoring web services, called Active Service Broker Architecture. In their model, an external process is responsible for the monitoring. It extends the more traditional service registry to provide management and quality control capabilities. The architecture also enables the collaboration between distributed monitoring agents at the service provider's sites and the centralized monitoring management at the service- broker site. It still has some issues regarding inadequate instrumentation of the sensors, and monitoring functions will be difficult to extend. [13]

Apostol et al. [12] have created a plugin for the Chrome web browser. It tracks what the user does on a website, and sends that data to an external database. It is capable of tracking general metrics like the loading times and external links. The back-end is implemented with php and a MySQL- database. The researchers propose the application to be extended to include more information about what the user is doing on the site, such as which pages he visits. They also propose switching to technology which allows for greater extensibility.

MOST4FIRE by Al-Hazmi et al. [9] provides a flexible monitoring system for federated cloud services (cloud services distributed over several providers). It can monitor several layers from infrastructure to services, but is rather cumbersome in use. Al-Hazmi et al. [8] have also created a monitoring service for the BonFIRE project. The service uses agents implemented in VMs to monitor the infrastructure.

Barbon et al. [14] describe a novel solution for monitoring distributed business processes in BPEL (Business Processing Expression Language) in a Java environment. They have separated the monitoring engine from the execution engine, which are running in parallel on the same application server. The monitors intercept the input/output messages that are received or sent by the processes. They are able to specify boolean, statistic and time-related properties to be monitored. Barbon et al. [14] also have instance monitors which can for example count the number of iterations that are executed in a given session, such as the number of times a user does certain actions. They can even issue alerts based on what is happening in the application [14].

All of the presented solutions are more or less individual solutions that work in very specific situations. None of them cater to what is needed in this thesis. Systems more suitable for the goals of this thesis are those with a complete measurement solution that adapts to the needs of the users.

2.9 Measurement frameworks

Mendonca and Basili [40, p. 484-485] define a good Measurement Framework (MF) as sound, complete, lean, and consistent: "An MF is sound when its metrics and measurement models are valid in the environment where they are used. An MF is complete when it measures everything that its users need to achieve their goals. An MF is lean when it measures what is needed and nothing else. An MF is consistent when its metrics are consistent with the user goals."

Offen and Jeffery [42] argue that a generic model-driven toolset framework is a good starting point for a measurement framework due to companies needing to minimize the cost and effort associated with software measurement. They continue that the framework should be robust, automatic, self-explanatory and easy to use. Furthermore, tedious and work-intensive measurement-related work practices like data collection should not overburden developers. [42]

Offen and Jeffery [42] also recognize that companies should not rely on a fixed set of ready-to-use tools because the amount of different measurement goals and types of measurable software and processes is so large. To solve this issue, they propose a framework called Squatter, which lets new tools to be added and new data sources or data repositories to be defined. [42]

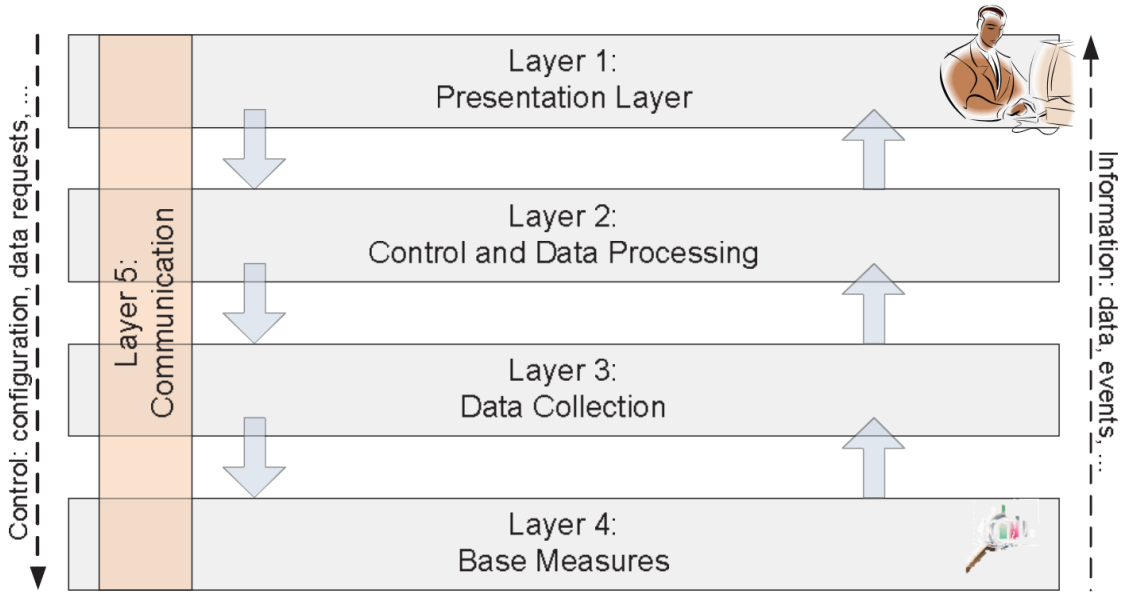

Figure 7: MFW high-level architecture. [34]

For a more general use, Kanstrén et al. [34] have proposed a five-layer architecture for monitoring a distributed system, as seen in Figure 7. The first layer of the monitoring framework (MFW), Presentation layer, shows the user the final metrics data. Layer 2, Control and Data Processing, is used to control the data collection and process the received data. In layer 3, Data Collection, the data is collected from one or more measurement probes. Layer 4, Base Measures, describes each

individual measurement probe which do the actual metrics gathering. Finally Layer 5, Communication, ties all the other layers together into one coherent system.

Becker et al. [15] describe a system, shown in Figure 8, for collecting and analyzing data for decision making in businesses. It is non-intrusive, allowing fully automatic capture of all data, and it supports a distributed development environment. The system is constructed of four layers: data capturing, data cleaning, data storing and data visualization. [15]

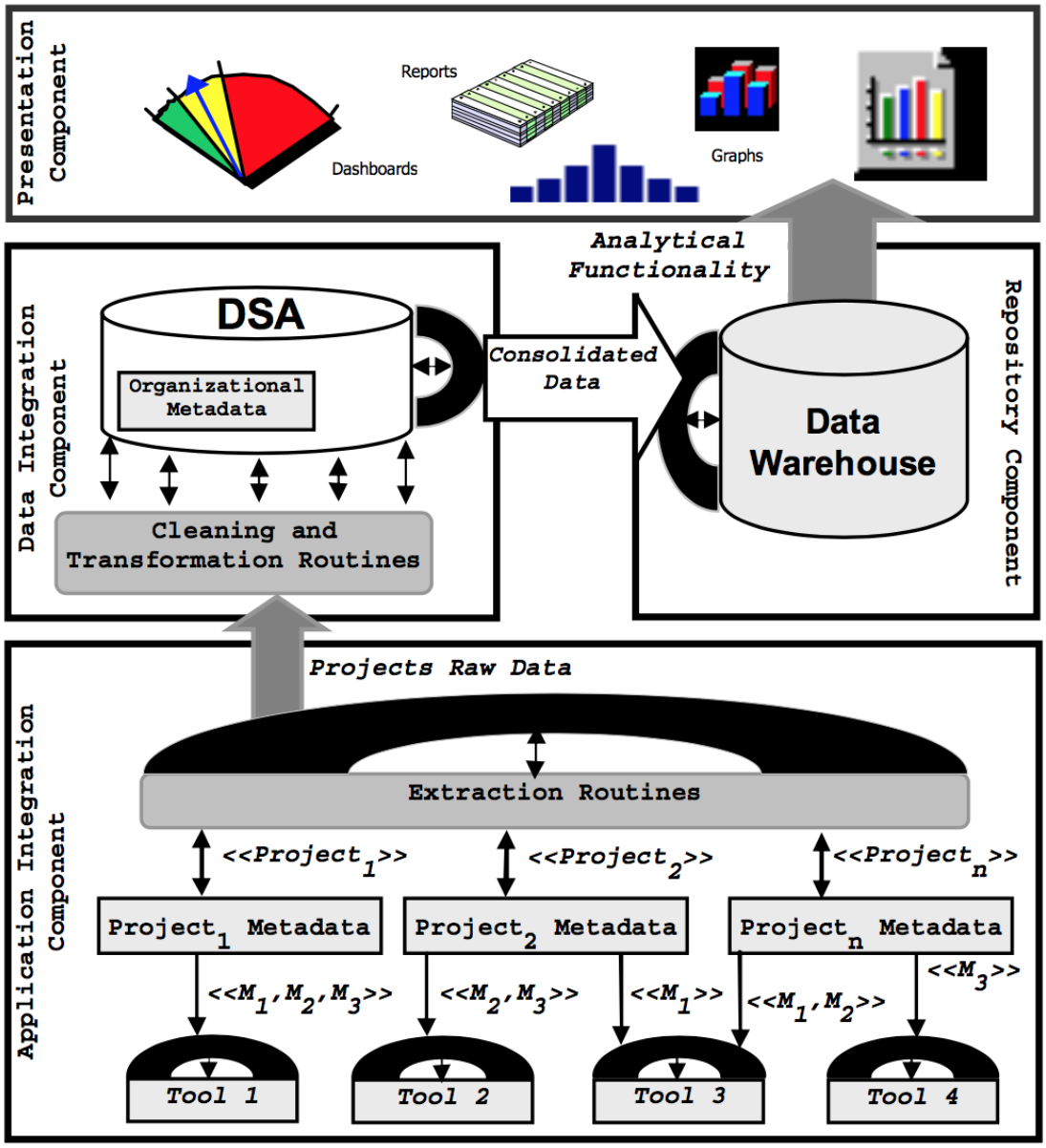

Figure 8: Data warehousing architecture. [15]

2.10 The analytics stack

Seufert [48] presents an analytics stack as a complete solution for collecting metrics, shown in Figure 9. It can be broken down into three component pieces: the back-end, the events library, and the front-end.

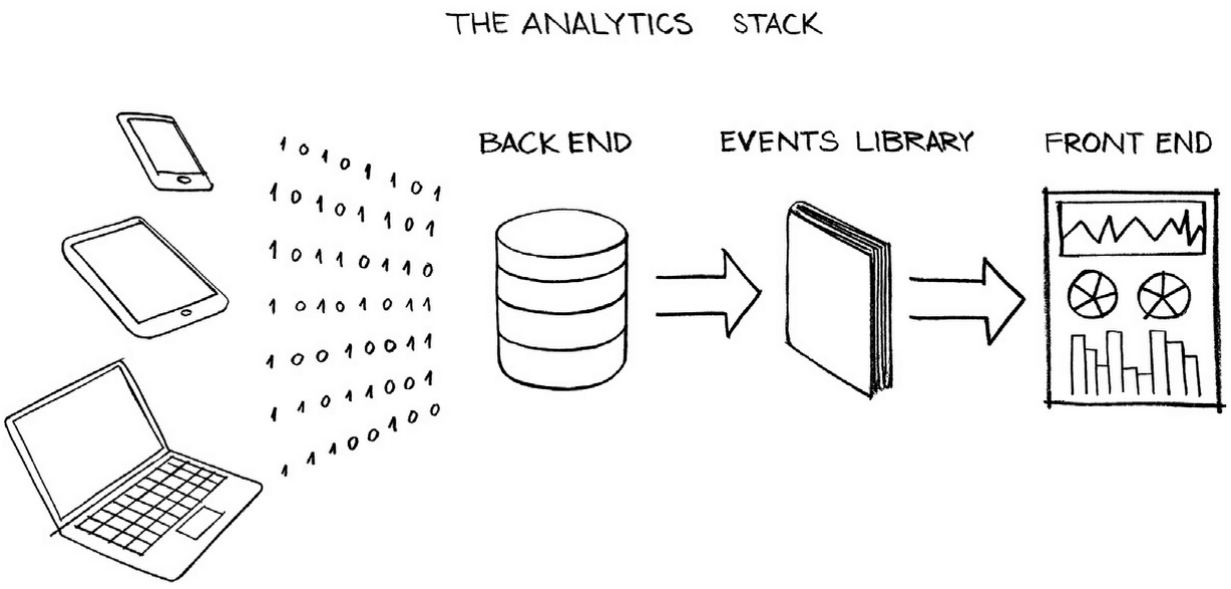

Figure 9: Illustration of the analytics stack. [48, p. 32]

The back-end is a storage mechanism that collects data. It can be implemented in any number of ways using any number of technologies. Traditionally back-ends have been built on relational databases such as MySQL or PostgreSQL, but the shift towards Big Data is favoring NoSQL- databases. [48]

The events library is meant to record and transmit data to the back-end. Seufert [48] suggests that it take the form of a discrete list of events and important data points that are meant to be tracked. Events libraries could be integrated into software clients, but as Big Data is becoming more important, events libraries are being replaced with logs that keep track of literally everything that happens in a product. However, Seufert still sees that events libraries will remain relevant in the future as well. [48]

The front-end is any software that retrieves and processes data stored in the back-end. Seufert [48] recognizes that the front-ends are usually provided by Third-party solutions, mentioning products such as Tableau, QuickView, and Greenplum which allow product teams to connect to data sets and almost instantly use them to build graphs and charts, segment data, and spot patterns. [48]

Some analytics stacks rely on ad-hoc querying and analysis in lieu of a front-end; an analyst will query data directly from the database and manipulate it using a statistical interface or desktop spreadsheet software. In other cases, a bespoke front-end is developed from scratch to fit the specific needs of product teams. [48]

Finally, Seufert [48, p. 32] acknowledges that there can be a big difference to the functioning of the components based on how they are developed and maintained: "distributing each component across three functional teams within an organization might disrupt the extent to which each piece communicates with the others". He also recommends to keep the analytics stack at the hands of a single entity in order to mitigate the problems arising from poor communication. [48]

3 Requirements

3.1 Introduction

This chapter will define the requirements for the metrics gathering tool. The requirements will dictate what the tool will do and how exactly it should work. The requirements will later be evaluated to check that the implementation fulfills all of them.

A requirement is a basis for any software. Sommerville [51] identifies a requirement as a statement that can range from a high-level abstract statement to a detailed mathematical function. Requirements can have various levels of classification. The IEEE standard for Systems and software engineering - requirements engineering [6] identifies a well-formed requirement as a statement that:

- can be verified,
- has to be met or possessed by a system to solve a stakeholder problem or to achieve a stakeholder objective,
- is qualified by measurable conditions and bounded by constraints, and
- defines the performance of the system when used by a specific stakeholder or the corresponding capability of the system, but not a capability of the user, operator, or other stakeholder.

A More detailed view of the requirements can be formed with a Software Requirements Specification (SRS). For a good SRS, the standard [6] recommends each requirement to posses nine characteristics. The measurement should be:

- Necessary - the requirement is applicable, essential and cannot be replaced by other requirements if removed.
- Implementation free - the requirement states what is required, not how it should be met, being implementation-independent.
- Unambiguous - the requirement can only be interpreted in one way, and it is stated clearly and easy to understand.
- Consistent - the requirement does not conflict with other requirements.
- Complete - the requirement is measurable and it sufficiently describes the capability and characteristics of the system.
- Singular - the requirement statement only defines one requirement.
- Traceable - the source and lower-tier requirements specification of the requirement can be identified such that the requirement traces to both its source and implementation.

- Verifiable - the requirement has the means to prove that the system satisfies the specified requirement, for example through measurements.

Requirements can be collected together in sets. This can help to avoid requirements changes and growth that could impact the cost, schedule or quality of the system. To prove that the individual requirements can function well together and provide feasible solutions for the system, a set of requirements must possess the following characteristics [6]:

- Complete - the set of requirements contains everything needed for the definition of the system being specified.
- Consistent - the set of requirements does not contain conflicting or duplicate requirements.
- Affordable - the complete set of requirements can be satisfied by a solution that is obtainable or feasible within the given constraint such as cost or schedule.
- Bounded - the set of requirements remains focused and does not move beyond what is needed to satisfy user needs.

Requirements can be divided into two types: user requirements and system requirements. *User requirements* are written for the users in natural language and diagrams to demonstrate the operational constraints of the system. User requirements define how the user interacts with the system and what the user should see as a response. *System requirements* define more thoroughly what should be implemented and concentrate on the system itself. It is often a structured document detailing each part of the system, its functions, services and operational constraints. [51]

System requirements are further detailed by functional and non-functional requirements. *Functional requirements* are specific, system-related statements that define the services that should be found in the system. These dictate how the system should handle different inputs and situations and what its response is. The definition and implementation of functional requirements is often based on non-functional requirements. [51]

Non-functional requirements are broader descriptions of how the system should be implemented and operate. These include for example system constraints such as reliability and response time or process requirements such as a specific development language or target system. It is often more critical to reach the non-functional system requirements; if they are not met, the system might be useless. [51]

Non-functional requirements can be classified in three categories. Product requirements specify how the system must behave, for example with regards to execution speed or reliability. Organizational requirements are devised from organizational policies and procedures such as the used process standards or the implementation requirements. External requirements arise from factors outside the organization, such as legislation or system interoperability. [51]

Sommerville [51] lists several metrics for specifying non-functional requirements, shown in Table 1. These metrics can be used to assess the non-functional requirements

and the success of their implementation in the final system. For example, the system can be checked for speed by measuring how many transactions per second it is able to process.

Table 1: Examples of metrics for specifying non-functional requirements. Adopted from [51]

Property	Measure
Speed	Processed transactions/second
Size	Mbytes
Ease of use	Training time, no. of help frames
Reliability	Mean time to failure, availability
Robustness	Time to restart after failure
Portability	Number of target systems

3.2 Target environment

To understand where and how the tool for gathering business metrics will be used, it is important to know what the target environment is like. The tool that is built in this thesis will be part of Codemate's new development environment. It has two main goals: To give the developers a simple and unified environment to work in, and to provide Codemate with easier ways to manage infrastructure usage and costs.

The environment allows developers to start new projects and request computing resources for them with minimal setup. Managers will be able to see detailed information about resource usage by different projects, which allows them to allocate the costs correctly between different projects. The applications that can be developed in the environment can be any web or mobile (back-end) services that use one or more of the Amazon Web Services (AWS). The applications are not strictly bound to AWS, however, as they can be built in a way that allows them to be moved to other cloud services.

Figure 10 shows the basic architecture of the environment. The environment is built on top of AWS and Kubernetes. The applications running at the top are separated from each other. Development will be done with Go-, JavaScript- and Ruby- languages. In addition to the basic architecture, the service uses several smaller components for functionalities, such as resource monitoring, which is implemented with InfluxDB time-series database [1] and Grafana [2] for visualization.

AWS [3] is a cloud platform built by Amazon. It is based on infrastructure resources which are shared dynamically between several users. Developers can access these resources and build their applications to take advantage of the elasticity, scalability and affordable cost structure of AWS. It offers a host of tools for the developers

[1] https://www.influxdata.com
[2] https://grafana.com
[3] https://aws.amazon.com/

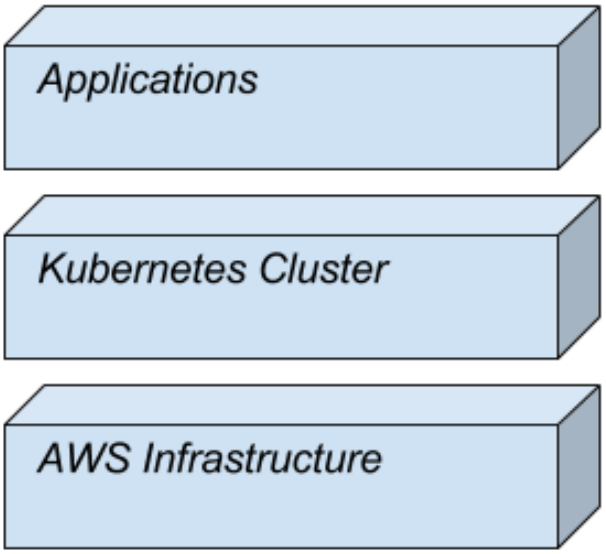

Figure 10: Architecture of the development environment.

which allow them to control the infrastructure of their services in a very detailed way. The usage of AWS is charged based on the duration, amount and size of the services used.

Kubernetes [4] is a container orchestration system designed to be used in cloud platforms. Separate applications can be organized in self-sustaining containers, such as Docker [5], and then given to Kubernetes to automatically run the applications. The benefit of Kubernetes comes from its ability to run several instances of individual containers, providing easy-to-use distribution and scalability. Kubernetes also brings better resource utilization and deployment density as well as better updatability of the services through rolling updates of the replica-set.

3.3 User requirements

3.3.1 General user requirements

The requirements for the system have been formed through discussions with the architects and software engineers at Codemate. The tool for gathering metrics will connect to the architecture presented in Figure 10 and it should provide the following three main functions:

- Capture business metrics from the applications;
- Store the metrics; and
- Visualize the metrics.

[4] https://kubernetes.io/
[5] https://www.docker.com/

The user requirements for the tool are defined through the two main user groups: application developers and system and product managers. Application developers will implement the gathering of the metrics in their applications including sending data to the database. The developers will also create the initial dashboards for the visualization tool. The system and product managers deal with the application from which business metrics are captured. They will use the visualization tool to analyze the data from their application and possibly manually create dashboards.

The installation of the system is handled by the developers and should be straightforward. Once the necessary measurements and points of insertion have been defined, the tool should require minimal setup to start to capture metrics; mainly only connecting the tool to the target application. To create the visualization dashboard(s), the developers should be able to automatically generate a functioning dashboard from simple definitions that take no more than few minutes to write.

When a measurement is captured by the application, the measurement should be sent to the database with the associated timestamp and identification. When data needs to be visualized, the product managers should be able to open their web browser, login to the service and see data about their system. The data should be fetched from the database and shown in graphs, tables and by means of other visualizations, organized by time.

The system should be easily modifiable. It should be possible to add additional types of measurements to it later if needed. Also the components of the system should be replaceable without having to modify the rest of the system. An example of this would be changing the visualization tool, which should not affect the database or metrics libraries. Finally, the system should be able to handle several applications while keeping the users and data separated.

3.3.2 Gathering metrics

To gather metrics from target applications, a subcomponent should be created. The metrics gathering should be separated from the individual applications while being easily accessible to them. Gathering the metrics should be simple for the developers both in terms of installation and maintenance.

The metrics will be gathered inside the individual applications. Because it is not known beforehand which metrics each application needs, this thesis aims to provide a robust and adaptive system for collecting several types of metrics. The different types of metrics that should be supported are:

- Counters - counts single values;
- Meters - tracks the occurrence of events with moving averages; and
- Histograms - samples dataset for distribution with mean and standard deviation.

To collect the metrics at runtime, a code library for creating and capturing the metrics should be chosen. The library should give developers easy access to methods with which they can collect and store metrics. The library is responsible for creating individual metrics and sending data to the database. Because the developers at

Codemate often use Ruby-, JavaScript- and Go- languages, the tool should support at least one of those.

3.3.3 Storing and visualizing metrics

The metrics will be stored in a database. The stored data must be timestamped, meaning that each stored value has a timestamp attached to it. The data must be query-able by time and the name of the metrics. This leads to the requirement that the database be a timeseries-database, because they are optimized to handle precisely the type of data used in this thesis.

The database should be able to handle data from several different applications. The data must be provided to the visualization tool. To minimize the bottlenecks in the system, and good scalability and availability, the database should run on Kubernetes with replication.

There are a few databases that would be suitable. The most prominent ones are InfluxDB, Graphite [6] and Prometheus [7]. They all store timestamped data and allow it to be queried. The differences come from the schema-less design of InfluxDB which allows it to also store strings on top of numerical values. Prometheus on the other hand includes the visualization tool by default. Graphite is the oldest of the three, offering the least functionalities.

With regards to the target environment, the database will be InfluxDB. It offers the best selection of functionalities, and is suited well to the current project. Codemate is already using InfluxDB on other parts of their development environment, so choosing InfluxDB also helps keep the tool in line with the other parts of the environment.

The metrics will be visualized using a data visualization tool. The tool should be able to read data from one or more databases and provide simple and clear representations of the desired metrics. The visualization should be understandable and usable also for the users of the final service.

There are few choices for the visualization tool, mainly Grafana and Prometheus. Prometheus offers a slightly better set of visualization tools, but requires its own database to be used. To ensure that the tool can be easily modifiable in the future, Grafana is better suited here. Codemate's development environment is already using Grafana and the developers and customers are familiar with it, so Grafana is the best option for the visualization tool also in this thesis.

3.4 System requirements

System requirements detail the user requirements given above. They provide clear and measurable requirements that can easily be verified. This section lists several requirements which the system must satisfy. First, functional requirements state what the system must be able to do. Non-functional requirements then detail how the system must perform and provide the necessary functions.

[6] https://graphiteapp.org/
[7] https://prometheus.io

Functional requirements

1. The tool must capture user-chosen metrics from the target environment.
2. The tool must capture at least the following types of metrics:
   (a) Counter;
   (b) Meter; and
   (c) Histogram.
3. The capturing of metrics must be simple and easy to use.
   (a) Adding the needed code to the project should not require more than a few lines of code or five minutes of time.
   (b) Adding a new measurement should not require more than a few lines of code.
4. The captured metrics must be sent to a database.
   (a) Sending measurements to the database should happen automatically at certain intervals.
5. The visualization must support at least the following graphical elements:
   (a) Graphs;
   (b) Tables;
   (c) Gauges; and
   (d) Individual measurements.
6. The dashboards must be generated from definitions.
   (a) The definition should include the layout as well as the queries from the database.
   (b) The generated dashboard must easily be installed to the visualization tool.

Non-functional requirements

7. The capturing of metrics must not affect the main system.
   (a) Initializing the capturing should not add a noticeable delay to the start of the application.
   (b) Capturing a metric should not noticeably slow the operation of the application.
   (c) Sending the metrics to the database should not have any noticeable effect to the usage of the application.
8. The metrics capture must be usable on Ruby, JavaScript or Go-lang.

9. If an external code library is used, it should be actively maintained.
    (a) The library should have received the latest updates no longer than three months earlier than the date of selection of the library.
    (b) The maintainers of the library should indicate that they will continue to maintain the library by showing activity in the issue reports in the past three months.
10. The database must be Influxdb.
11. The visualization must be done using Grafana.

4 Design

4.1 Introduction

The design and architecture of the metrics gathering tool will be based on the previous research outlined in Chapter 2 and on the requirements presented in Chapter 3. In order to clearly present the architecture, the 4+1 View Model by Kruchten [36] will be used. The 4+1 View Model describes software architecture with five different views as shown in Figure 11.

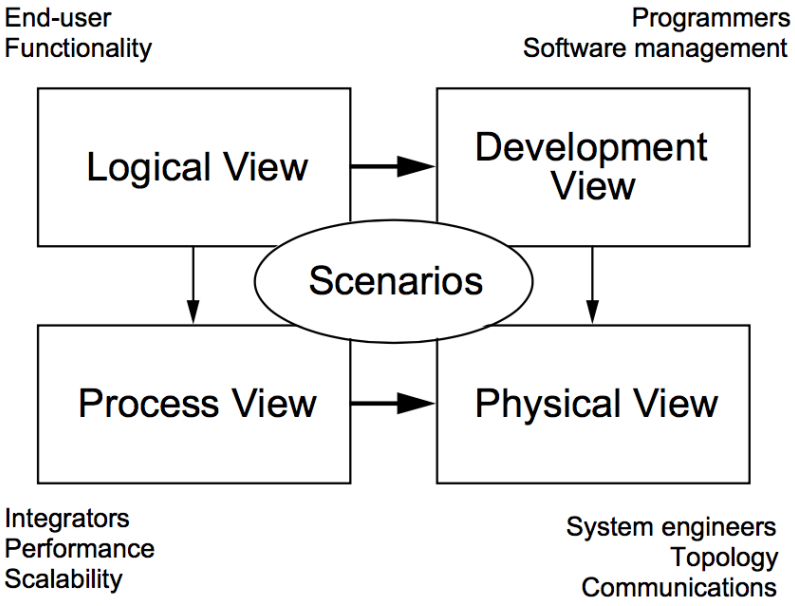

Figure 11: The 4+1 View Model. [36]

Logical view describes the system from the point of view of the end-user. It maps the desired functionality into components that provide that functionality. The logical view is the "big picture" of the system and addresses the functional requirements of the system.

Process view helps the designers of the system. It is meant to explain the system interaction and how individual subsystems can communicate with each other. The process view addresses some of the non-functional requirements and how the individual processes of the system can be built, taking into account replication, distribution and availability.

Development view is meant to give guidelines to the developers of the system. It is mainly concerned with the act of development considering things such as ease of development, software management and constraints imposed by the programming language. The development view shows how the individual modules of the system are organized and what dependencies they have.

Physical view is directed at the system engineers and administrators. It deals with the physical properties of the system such as the devices on which the system

should run, mapping the properties identified in the other views onto physical computing nodes. It deals with most of the non-functional requirements such as system reliability, performance and scalability.

Finally, **Scenarios** describe how the other views come together to form one coherent system. The scenarios view is described as "+1" because it is redundant with the other ones. In the scope of this thesis, the User requirements devised in Section 3.3 essentially serve the same function. As such, the scenarios will not be discussed further.

The following sections will go over the design of the tool for gathering business metrics. The design follows the 4+1 View Model, starting from the logical, user-centric view and moving towards the physical view.

4.2 Logical view

To meet the user requirements for the system and provide all the desired main functionalities - capturing, storing and visualizing business metrics - the tool will have a modular structure presented in Figure 12. The main components form individual parts of the service that communicate with each other via general interfaces. The dashboard generation is a separate component which connects to the visualization component.

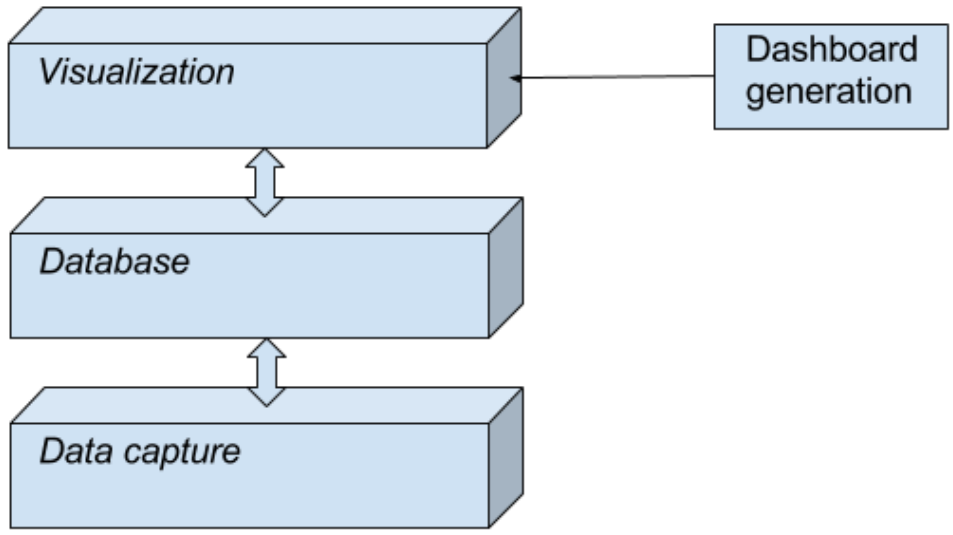

Figure 12: Architecture of the developed system.

The architecture follows the SBA- principle and the design builds on the existing literature presented in Chapter 2. It implements the proposal of different layers within the MFW by Kanstrén et al. [34] (Figure 7). This abstraction is further detailed with help of the data warehousing architecture (Figure 8) by Becker et al. [15]. By imagining the database as the back-end and the visualization tool as the front-end

of the application, the analytics stack by Seufert [48] presented in Chapter 2.10 can be used to guide the design of the user experience for the end-users.

As is the case in the presented literature, the chosen architecture clearly separates the whole system into subsystems. This requires them to communicate with each other by means of general interfaces or APIs. Each subsystem functions on its own without requiring the other subsystems, except to provide system-wide functions. The subsystems only communicate with each other when necessary, and the visualization and data capturing subsystems communicate only through the database.

The dashboard generation for the visualization tool will be a separate component which receives its input from the developer. It functions on its own without requiring any other systems. The output of the dashboard generator is a json-file which can be manually exported to the visualization tool.

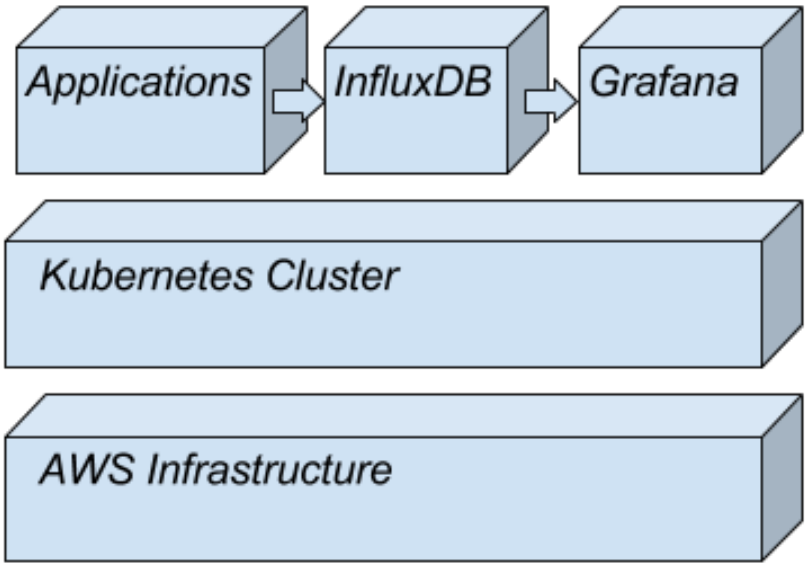

Figure 13: Final architecture of the developed system.

Figure 13 shows the final architecture of the Codemate's system. The whole system is run in AWS on top of Kubernetes. Applications, database and the visualization tool have their own Kubernetes namespaces which separate their instances. The application sends data to the database, and the visualization tool uses that data.

4.3 Process view

4.3.1 Capturing metrics

As discussed in Chapter 2, there are two main ways to capture metrics from an application: interception and instrumentation. Because interception monitors the process from the outside, it is not suitable for capturing business metrics, which are generally generated inside the application itself. Therefore the metrics will be

captured instrumentally with probes implemented directly into the application code. To minimize the intrusion to the target application, a separate code-library will be used. The library should be chosen such that it does not affect the performance of the target application.

When the code execution reaches the point where a metric should be captured, the code will store the metric in the local memory of the server. The stored data will include the metric as well as meta-data such as information about the metric. The captured data will periodically be sent over the network to the database. Ideally, the database is located as close to the target application as possible, in order to minimize the network latency and keep the network secure. The communication protocol will depend on the database, which the library must support.

4.3.2 Storing the metrics

As per the requirements discussed in Chapter 3, the metrics will be stored in the InfluxDB database. It is an open-source time-series database that is "meant to be used as a backing store for any use case involving large amounts of timestamped data", including application metrics and real-time analytics. It is highly available with a REST interface. [3] Through the RESTful API it can easily be used by both the library capturing the metrics and the visualization tool that presents the measurements.

The general schema for storing the metrics in the database is shown in Listing 1. The "name" parameter stores the name of the measurement. The "columns" array holds the name of the different fields that contain the measured values. Time and one field are mandatory for all measurements. Additional fields and tags are optional. Tags can be used for indexing the measurements for faster queries. "values" is a list of the actual values, as dictated by "columns".

```
{
  "name": "measurement.name",
  "columns": [
      "time",
      field,
      [fields,]
      [tags,]
  ],
  "values": [],
}
```

Listing 1: "General schema for the database"

Appendices A - D show the actual schemas for Counters, Meters, Histograms and Timers. They follow the general schema definition, adding more fields as required. The "count" in Counter holds a simple value of the counter. The Meter also stores one-minute, five-minute and fifteen-minute moving averages as well as the mean-rate of the "count" value. Histogram tracks several measurements derived from the count. Finally, Timer is a combination of the Meter and Histogram.

To ensure good scalability, availability and ease of deployment, the database will be run on Kubernetes. It is "an open-source system for automating deployment, scaling, and management of containerized applications" [4]. Containers are self-contained and isolated runnable packages. They guarantee that a software always runs in the same way regardless of where it is deployed. One of the world's leading container platforms is Docker, and it will also be used in the tool created in this thesis. [1]

The database will run continuously, waiting for requests through its RESTful API. When it receives data from the application, it will store that data in its databases. When data is requested by the visualization tool, the database will serve the requested data to authorized users. Each application will have a datatable of their own, which is only accessible to authorized users of that application.

4.3.3 Visualizing the metrics

Because Codemate is already using Grafana, and it is a functional requirement, the metrics are visualized with it as well. Grafana is an open-source visualization tool that supports several different databases and provides many different visualization tools. It allows the creation, exploring and sharing of dashboards that show data queried from one or more of the supported databases. [2]

The main visualization tools supported by Grafana are:

- Graph - a 2D graph with time at x-axis;
- Table - a table with time as the first column;
- Heatmap - a map of values with colors;
- List - a list of Dashboards, alerts or plugins; and
- Single stat - a single value or a gauge with alert thresholds.

Grafana provides a web-service through which the users can use the visualization tool. The users can connect to the interface with their browser at a specified web address and login with their credentials. Each user is only given access to data that they are allowed to view. Grafana has a user-management system with which different users and applications can be separated.

4.4 Development view

The system will be implemented by a single developer, the author. The development process does not therefore need any distinct formalities. The project is divided in two different repositories: one for the main project containing the scripts for starting the main service, and one for the JavaScript library. The components will be built in a lean fashion.

Because the main back-end languages in use at Codemate are Ruby, JavaScript and Go, one of these will be chosen for the implementation language for the scripts

and the dashboard generation. The author has decided to use JavaScript with Node.js because he is most familiar with it, and this tool can be tested on a project that is using JavaScript.

4.5 Physical view

As mentioned in Chapter 4.3, the data storing and visualization components will be using Kubernetes as their backbone. Kubernetes on the other hand will be run on AWS to keep the system coherent with the rest of Codemate's development environments.

AWS includes a wide variety of services to accommodate a host of user needs. The tool implemented in this thesis only needs a few of them. The Kubernetes clusters will be run on Amazon Elastic Compute Cloud (EC2) instances. For the initial implementation, data from the user applications into the database will be routed through Elastic Load Balancer (ELB) for proper load balancing. In addition, EC2 Security groups will be activated to only allow traffic from known hosts such as the applications and the individual components of the system.

5 Implementation

5.1 Implementation method

The data capturing tool designed in Chapter 4 is developed in two stages. The first Stage implements a minimum viable product (MVP) that works in a local environment. The second Stage extends the implementation to a full-blown web-based application.

Each development stage includes the development of the whole system and its four subsystems. The main technological difference between Stages 1 and 2 is that Stage 1 uses Minikube for simulating a cloud environment in a local development environment (Macbook Pro- laptop), while Stage 2 makes use of the AWS services.

Minikube [8] is a tool which allows for running Kubernetes locally on a single node. Essentially, it replicates a cloud environment such that applications running on it think that they are in the cloud. Minikube runs in a virtual machine which for this thesis is Virtual Box [9].

As the main components in each stage are the same, they will be presented within Stage 1 in Sections 5.2 - 5.5. Section 5.6 will present the Stage 2 and how the components have been adapted for it.

5.2 Metrics capturing

The metrics are captured using a popular metrics library for Node.js, metrics by Mike Ihbe [10]. It is based on the widely popular metrics library for Java by Coda Hale. The metrics- library allows the capture of several types of metrics such as counters, histograms and timers.

To send the captured data to the database and extend the metrics- library's functionality, the tool will use another library by Brandon Hamilton, called node-metrics-influxdb [11]. The library already imports the metrics- library, so only one library needs to be added to the tool. Hamilton's library also implements gauge, which is missing from Ihbe's metrics- library. The library however has some flaws when it comes to multi-threading and connecting to the database, so the author has created a fork (new version) of the library.

The main addition in the forked library is the replacing of custom-made HTTP (Hypertext Transfer Protocol)-connection functions with the official library for connecting to the InfluxDB. Also some bug fixes and small functionality changes have been made which better support the needs of this tool. After the changes, the forked library was published as a new npm- package called metrics-influx which is public and available for everyone to use.

The forked library works by first defining a new measurement to capture. It is then registered for monitoring. Once the program is running, the code will wait for additions to the measurement. To send the captured data to a database, the

[8] https://github.com/kubernetes/minikube
[9] https://www.virtualbox.org/
[10] https://github.com/mikejihbe/metrics
[11] https://github.com/brandonhamilton/node-metrics-influxdb

database must be connected to the application and an interval defined. Once started, the code will periodically send all the newly captured metrics to the database via a RESTful HTTP-call.

Listing 2 shows a minimal example of how the library can be used in the application. First, the library must be imported to the code. Next, a reporter is created which will send data to the database every 5 seconds (5000ms). Options for the reporter include the url for the database as well as username and password, provided in json-form. Finally, the different metrics are created, and a single value reported to each.

```
var InfluxMetrics = require('metrics-influx');

var reporter = new InfluxMetrics.Reporter(options);
setInterval(reporter.report.bind(reporter, true), 5000);

var c1 = new InfluxMetrics.Counter();
reporter.addMetric('measurement.count', c1);
c1.inc();

var g = new InfluxMetrics.Gauge();
reporter.addMetric('measurement.gauge', g);
g.set(50);

var m = new InfluxMetrics.Meter();
reporter.addMetric('measurement.meter', m);
m.mark();

var h = new InfluxMetrics.Histogram();
reporter.addMetric('measurement.histogram', h);
h.update(100);

var t = new InfluxMetrics.Timer();
reporter.addMetric('measurement.timer', t);
t.update(200);
```

Listing 2: Minimal usage example of the library

When a measurement is made, the library is responsible for calculating all the necessary measures that are stored in the database. This puts some extra burden on the measuring probe, making it an instrumented probe with analysis. On the other hand, the database only needs to store the data instead of modifying it, which makes it more efficient and able to handle requests from several applications. The same applies for the visualization tool.

5.3 Storing the metrics

The metrics are stored in the InfluxDB database which is run on top of Kubernetes. In the local environment, the Kubernetes cluster is run on Minikube. By running the

database on top of Kubernetes, it is more robust for service breaks and also ready for scaling.

To install InfluxDB, an installation script was created, shown in Listing 3. The script downloads an image of the InfluxDB and starts it on Kubernetes in port 8086. The script then exposes the instance and creates a public IP address for it. The type of the instance is NodePort, which has no loadbalancing. This is used in the local environment only, as loadbalancing is not available on Minikube. Once the InfluxDB can be reached over the internet, the script connects to it via curl and creates an admin- user and a new database. The database is then operational and ready to receive data and respond to queries.

```
kubectl run --image=influxdb influxdb --port=8086 --env="
   INFLUXDB_HTTP_AUTH_ENABLED=true"
kubectl expose deployment influxdb --type=NodePort --port
   =8086
echo Getting IP address for the container
ipDatabase=$(minikube service influxdb --url)
echo Creating admin user
curl -s -S -i -XPOST $ipDatabase/query --data-urlencode "
   q=CREATE USER admin WITH PASSWORD 'password' WITH ALL
   PRIVILEGES" >/dev/null
echo Creating database data
curl -s -S -i -XPOST $ipDatabase/query -u admin:password
   --data-urlencode "q=CREATE DATABASE data" >/dev/null
```

Listing 3: InfluxDB installation script

5.4 Visualizing the metrics

The metrics are visualized using Grafana which is run on Kubernetes. Installing Grafana first requires the use of an installation script shown in Listing 4, and then configuring it by creating users and adding the database(s) to it. The dashboards are created manually or by the generation scripts.

```
kubectl run --image=grafana/grafana grafana --env="
   GF_SECURITY_ADMIN_PASSWORD=password"
kubectl expose deployment grafana --type=NodePort --port=
   3000
echo Getting IP address for the container
ipGrafana=$(minikube service grafana --url)
echo IP: $ipGrafana
```

Listing 4: Grafana installation script

The installation script first downloads a runnable image of Grafana and then starts it on Kubernetes with a set password for the admin user. The instance is then exposed to the localhost as a NodePort in port 3000, again as a substitute to a

loadbalancer in the local environment. The script finally fetches the ip-address of the instance and prints it out to the console.

Appendice E shows images of the Grafana instance. To use the service, the user must first login to the service. At first, only an admin account exists, but he can create new users. Once logged in, the admin can continue installation by adding a new datasource. To add a new datasource with the type InfluxDB, the following must be specified:

- Name - the name of the datasource;
- Type - the type of the datasource (database);
- Url - the url of the database, including the protocol and port number;
- Access - the type of access for the database: proxy or direct;
- Http Auth - options for configuring the optional http authentication;
- InfluxDB details - the database name, username and password; and
- Default group by time - optional value for grouping fetched values by time.

With the database configured, dashboards can be added to Grafana. There are three ways to add a dashboard: create it manually, import or paste a json-file with the dashboard configurations from the computer, or import a dashboard by id from grafana.com- website. The created dashboards can be modified at any time. The modifications include for example:

- Title;
- Dimensions (width, height);
- Queries for data;
- Axes (scale, y- and x-axis min and max);
- Legend;
- Draw modes (bars, lines, points);
- Alerts; and
- Time range.

To get actual data, queries must be specified individually for each visualization. This tool uses InfluxDB, so the queries are made with an SQL-like language. An example of a query for all the counter values could be:

SELECT "count" FROM "measurement.counter"

5.5 Generating the visualization dashboard

To generate the dashboard for Grafana, a Go-program was created along with template- and Definitions-files. The program takes the definitions for the desired dashboard and applies them to the template-file, outputting a JavaScript Object Notation (JSON)-file. The outputted json-file can then be manually imported to Grafana as a functional Dashboard.

The definitions-file is a json-file containing all the definitions needed to generate a dashboard. By modifying only this one file, any type of dashboard can be created. The definitions-file contains the specific information for that dashboard such as the dashboard name and time range, the definitions for different data types which are graphs, tables, gauges and single values, and the definitions for Grafana template variables.

The template-file is a json-file based on the structure of the file that Grafana expects for importing a dashboard into it. The file contains general information about the dashboard such as its name, requirements and time options to show. It describes all the different rows and panels that exist on the dashboard. Several Go-template variables are inserted within the template at positions where data from the definitions file is needed. There are also loops which allow the addition of 0-n of each panels, as defined in the definitions-file.

The Go-program brings together the template- and definition-files. Once the program is built, it can be executed from the command line. The program starts by reading the definitions-file and checking that it contains all the required data. It then saves the variables into structs and reorganizes them ready for the template. The variables are then passed to the template-file and a new json-file generated, ready to be imported to Grafana.

5.6 Moving to Cloud

With the data capturing tool working in a local environment, it was next moved to the cloud. The main difference to the local version is that Minikube was replaced with AWS so that Kubernetes is running on actual server nodes instead of virtualized ones. These server instances are assigned individual IP addresses so that they can be reached over the internet. Because of this, a security policy needs to be put in place to control who and from where can access the tool. Finally, to keep the tool accessible at all times, load balancers are implemented to even the load between individual server instances.

The specific architecture of the cloud is shown in Figure 14. The basis is formed by the Amazon Virtual Private Cloud (VPC) which provides an isolated section of the AWS for this service. It has an IP-address range which allows the creation of subnets inside the VPC. The service has two subnets: virtual and public. The public subnet contains the load balancer and the host where external clients can connect to. The private subnet contains the Kubernetes master and nodes and manages the automatic scaling of the application. [19]

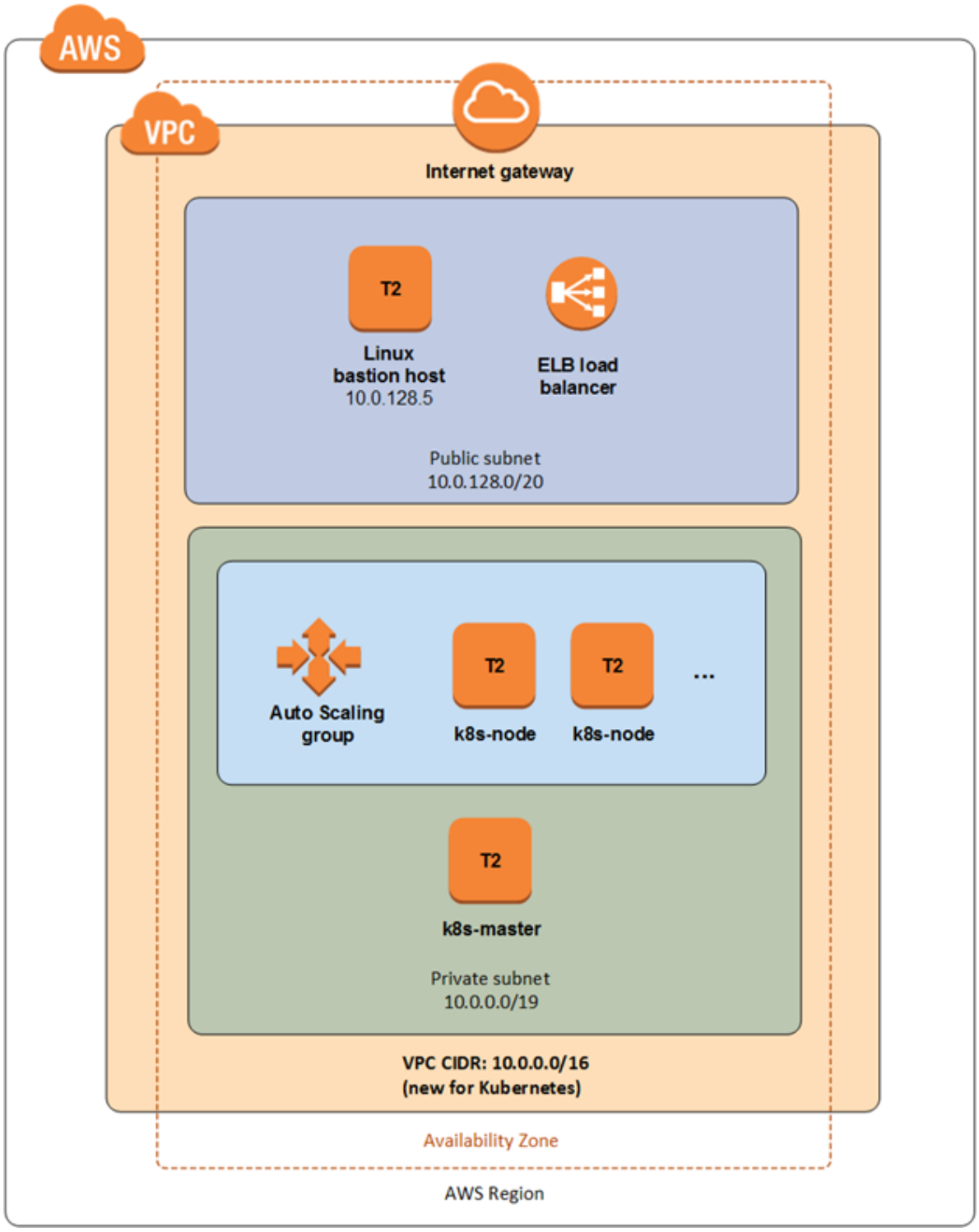

Figure 14: The implemented AWS Architecture. [19]

The actual service is provided by Elastic Cloud Compute (EC2) and Elastic Block Store (EBS). Load balancing is done with Elastic Load Balancer. EBS is a persistent block-level storage system which works together with EC2. EBS automatically replicates each volume within the same availability zone (such as Europe-Central: Frankfurt), offering high availability and durability. [19]

To install the basic infrastructure along with Kubernetes on AWS, a pre-made installation template by Heptio [19] was used. It allows one to configure the instances easily after which it takes care of installing them. The configuration used was:

- Availability zone: eu-west-1a;
- Admin ingress location: 0.0.0.0/0;
- Node capacity: 2;
- Intance type: t2.medium;
- Disk Size: 40GB; and
- Intance type (Bastion Host): t2.micro.

From this point on the installation and configuration of the system is similar to the local environment. Once an ssh-connection to the Kubernetes Master server is established, services for InfluxDB and Grafana can be started and IP addresses for them discovered. To establish an ssh-connection to the master node, the command shown in Listing 5 can be executed in the terminal.

```
SSH_KEY="[keyName].pem";
 ssh  -i $SSH_KEY -A -L8080:localhost:8080
  -o ProxyCommand="ssh -i \"${SSH_KEY}\" ubuntu@[publicIP]
     nc %h %p" ubuntu@[privateIP]
```

Listing 5: ssh-connection to the server

The installation of InfluxDB and Grafana are done in the same manner as for the local environment. The only difference is that now the services are exposed as LoadBalancers instead of Nodeports. To get the IP addresses of the services, a direct query to Kubernetes must be made and the output grepped, as opposed to receiving it directly from Minikube. Appendices G and H show the modified installation scripts.

6 Evaluation

6.1 The tool

The tool implemented in this thesis is fully functional and working as intended. This section will describe how the tool works and how it can be installed and Section 6.2 discusses the scalability and fault-tolerance of the tool. Section 6.3 evaluates the tool against the requirements formed in Chapter 3. Section 6.4 will show how the tool was used in a real production environment.

The installation of the system is planned to be straightforward with the scripts written for this thesis. First, a Kubernetes stack must be started on AWS (or any other cloud platform). Once Kubernetes is running, InfluxDB and Grafana can be installed on it with the scripts shown in Appendices G and H. When finished, the scripts print out the external IP-addresses for the instances which allow users to connect to them.

With the system installed, the database is waiting for data to be added to it. To insert data, a Node.js- library was implemented. It can be installed to the target system with the npm package manager. The library is then imported to the code following the standard npm package procedure, and configured with only a few lines of code. The configuration includes the options for connecting to the database. Listing 2 shows a minimal usage example of the library. Once the library is configured and the service started, measurements are periodically sent to the database.

To visualize the data, Grafana provides graphs, tables and figures. They can be configured in a straightforward manner to show any data in the database. Grafana can be reached with a regular web browser at the address given during the installation. Users can login to the Grafana service and see and query data available to them.

The data shown in Grafana can be organized into Dashboards, which can be created manually inside Grafana, or generated automatically with a generation script that was created alongside this thesis. The dashboard generation script generates the dashboards from a definitions-file where the developer can specify what data to show and where to show it on the dashboard. The script was created with Go-lang. To minimize the effort to build new dashboards, the script is an executable file that, once run, creates a json-file from a Grafana dashboard template- file. To install the dashboard to Grafana, the generated json-file only needs to be imported in it.

6.2 Scalability and fault-tolerance

The tool for capturing and visualizing business metrics has been created with scalability and fault-tolerance in mind. It is highly configurable and can be deployed in several ways and sizes. In essence, the scalability of the tool depends on two main things: The point at which captured data is processed, and the infrastructure on which the tool is deployed.

The data that the tool captures is meant to be analyzed through the visualizations in the visualization component. As such, the data can be processed into the correct form by all the different components. The metrics library which captures the metrics

can analyze and do the calculations on the measures immediately when capturing the measurements and before sending them to the database. Alternatively, the raw captured data could be sent to the database, and this data analyzed either by the database or the visualization tool when the data is queried. The current solution relies on the metrics capture library to do most of the analysis, and only small parts of it are left for the database, which it performs for each individual query as needed.

The service could also suffer from sudden increase in load. This could come from one or more applications which suddenly are generating more data, for example due to sudden increase in users. There are three key points where the service could break: the target application, the network between the application and the measurement database, and the database.

To handle increased load, the application and database instances can be scaled up. Alternatively, load could be moved from one component to the other; for example by letting the target application send raw data to the database instead of processing it first. To reduce network traffic and database requests, the measurements could be batched into bigger chunks which are sent to the database with a delay, buffering the measurements inside the application.

The tool can be deployed on almost any infrastructure which supports Kubernetes. The environment could have a simple server with virtualized nodes, or include several servers across different data centers. The currently deployed implementation only works within a single Availability Zone in AWS. This is subject to threats affecting the Availability Zone as well as the whole data center, such as hardware malfunctions, power losses or network disruptions. While such large outages and breaks in the operations of the databases are rare, they can happen. The only way to protect the service from them is to distribute it across several different Availability Zones and even data centers, preferably as far away from each other geographically as possible.

For this implementation, the separation could be achieved by running separate instances in different Availability Zones within the same data center. This would require the databases to replicate each other periodically. A common load balancer would also be needed to distribute the load between the different Availability Zones. To protect the service from outages of whole data centers, different data centers could be used. This also requires running several separate instances of the tool and database which need to be consistent. As such, managing the infrastructure would get more complicated than is presented here.

6.3 Requirements verification

6.3.1 User requirements

To verify that the implemented tool adheres to the specification and requirements outlined in Chapter 3, this section goes over each of the requirements and checks that all of them have been correctly taken into account. User requirements are evaluated in this section, and system requirements in Section 6.3.2.

The tool follows the general user requirements as it implements all three main functions: capturing, storing and visualizing business metrics. It also provides the

dashboard generation for Grafana. Both user groups, the developers and the product managers, have been taken into account by providing the functionality they need.

All main parts of the system function well together by using the http-interface and not requiring any additional data modification. This provides simple steps for capturing measurements from an application through code insertion only, and the dashboards can easily be generated by modifying only a single file. The visualization tool aims to be simple to understand and use. Lastly, the components of the system are easily replaceable because they are separated and use common interfaces. This satisfies the requirement for easy modifiability.

The tool can gather more types of metrics than specified in the requirements, including Counters, Gauges, Histograms and Timers. It is robust and can handle several different use cases and target applications. The tool provides a code library for Node.js for capturing the metrics.

All the captured data is timestamped and stored in a timeseries-database, InfluxDB. Data from several applications can be visualized with Grafana. They are both run on Kubernetes for fault-tolerance and replication.

6.3.2 System requirements

Table 2 shows all the system requirements presented in Chapter 3 and how they were fulfilled. The first functional requirement states that the tool must capture user-chosen metrics from the target environment. This means that the user must be able to choose what data and from which part of their application is gathered. This requirement is satisfied through the code library, which allows the capturing of metrics from any part of target application, letting the developer choose what and where to capture.

Requirements 2 and 3 deal with what kind of data and how specifically is captured. The tool can track simple Counters that count the number of something happening, Meters which also automatically keep track of the moving average for a counter number, and Histograms which provide statistical data about measurements. To capture the metrics, the developer only needs to import the metrics library, configure it with the database details, and then create the individual measurements. These steps are planned to require minimal effort from the developer through the use of simple, standardized steps and only a few lines of code. Requirements 2 and 3 are therefore both met.

The metrics code library provides functions for sending the measurements to the database either manually when specified by the developer or automatically at desired intervals. The measurements are then permanently stored in the InfluxDB database, which satisfies requirement 4. The visualization tool uses the measurements in the database and visualizes them using graphs, tables, gauges, and other visualization tools the developer chooses to use. Dashboards used in the visualization tool can be constructed either manually within the tool or by generating them automatically using the generation tool provided. Requirements 5 and 6 are therefore met.

Requirement 7 specifies that the capturing and storing of measurements should not affect the target system. The initialization of the metrics capturing is performed

with only few lines of code when the server is started, which has no effect on the performance as the server is started as fast as before. To capture a metric, a simple function call is made and the function executed, which has no noticeable delay for the user, which was tested manually several times. Finally, the measurements can be sent periodically to the database by executing the code in a parallel thread, which does not block the main thread. Requirement 7 is therefore satisfied.

Requirements 8-11 deal with specific implementation constraints. The code library used to capture metrics is made for Node.js and was developed by the author. The library however is based on a similar library of another user, and uses a third library for actually capturing the metrics and a fourth to send the metrics to the database. A Complex library structure such as this is not uncommon in Node.js but it can create problems with regards to maintainability in the long run. All the libraries are however more or less actively maintained alleviating the risk of one or more of them becoming outdated. Each library has had development activity in the last three months at the time of writing (May 2017). Finally, the database is InfluxDB and visualization tool is Grafana, which satisfies the last requirements.

Table 2: Requirements and how they were satisfied

	Requirement	**Method**
1.	The tool must capture user-chosen metrics from the target environment	Possible with the code-library
2.	The tool must capture at least Counter, Meter and Histogram types	The code-library allows the capturing of Counters, Meters, Histograms and more
3.	The capturing of metrics must be simple and easy to use	Only a single code-library needs to be imported
3.(a)	Adding the needed code to the project should not require more than a few lines of code or five minutes of time	At most 3 lines of code are needed
3.(b)	Adding a new measurement should not require more than a few lines of code	2 lines of code are required, plus one per each capture point
4.	The captured metrics must be sent to a database	The metrics are sent to the InfluxDB database
4.(a)	Sending measurements to the database should happen automatically at certain intervals	Possible by calling the report-function periodically
5.	The visualization must support Graphs, Tables, Gauges and individual measurements	Grafana supports all of these, and more
6.	The dashboard must be generated from definitions	The created Go-program allows this
6.(a)	The definition should include the layout as well as the queries from the database	Layout, queries, template parameters and more can be defined
6.(b)	The generated dashboard must easily be installed to the visualization tool	Grafana allows simple upload of json-files

	Requirement	Method
7.	The capturing of metrics must not affect the main system	The capturing has negligent performance impact on the system
7.(a)	Initializing the capturing should not add a noticeable delay to the start of the application	Starting the application is as fast as without the monitoring tool
7.(b).	Capturing of metrics should not noticeably slow the operation of the application	The operations remain fast
7.(c).	Sending the metrics to the database should not have any noticeable effect to the usage of the application	The application remains responsive
8.	The metrics capture must be usable on Ruby, JavaScript or Go-lang	The code-library supports JavaScript (Node.js)
9.	If an external code library is used, it should be actively maintained	All the libraries used have received regular updates
9.(a)	The library should have received the latest updates no longer than three months earlier than the date of selection of the library	All the libraries have been updated since February 2017
9.(b)	The maintainers of the library should indicate that they will continue to maintain the library by showing activity in the issue reports in the past three months	Github issue- pages are active for all libraries
10.	The database must be InfluxDB	The database is InfluxDB
11.	The visualization must be done using Grafana	The visualization is done with Grafana

6.4 Actual usage

The tool for capturing business metrics from cloud applications has been developed in this thesis to meet the needs of Codemate and its clients. It is therefore necessary to verify the functioning of the tool in a real production application. To do this, the tool was attached to the production server of one of Codemate's clients. The application contains a back-end server and two front-ends with different functionality: an iOS/android application for regular users and a web-based tool for management. The server is run on AWS and is built with Node.js.

The desired measurements that should be captured from the system are:

- How often do users login?
- How long does it take for the server to send a response when logging in?

These measurements can be extended to form more detailed business measurements (adapted from [44]):

- What is the average monthly growth rate for the service?
- When is the user activity the highest during a day or a week?
- What functions consume the most resources?

The desired data was easily captured with the implementation created in this thesis. First, a new Stack with Kubernetes was created for the InfluxDB and Grafana. Then, the code library was added to the code of the target application and pushed to the production server. Once the server was restarted, it started to send measurements to the database.

Figure 15 and Appendix F show dashboard items with actual measurements from the production system for login. The two numbers in Figure F1 show the total amount of logins in the system for the current day and the day before. It is implemented as a counter which is reset at midnight. The numbers shown are the maximum values of the counter for each day.

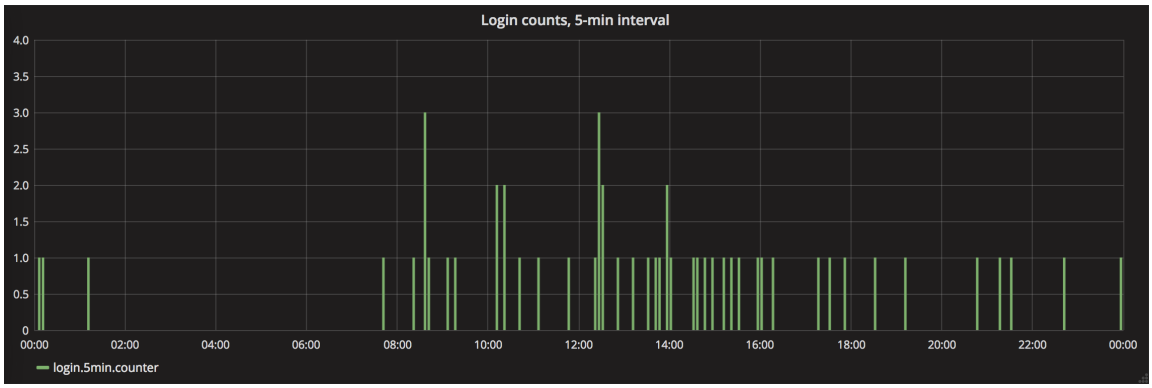

Figure 15: Graph with one-minute moving weighted average.

The login counts as 5-minute sum in Figure 15 shows how many logins have happened within each 5-minute period. To get a more detailed data of the logins, the table in Figure F2 shows each login separately. The table shows the exact time and daily count for each login as well as what the timer's minimum, maximum and mean values are.

The login min & max & mean- graphs in Figure F3 show the data from a timer in the login- function. The timer saves the time it takes for the server to process the login request, counting the time between the moment it receives a request from the client and the moment the response is sent. The graph shows the minimum, maximum and mean for the timer. Figure F4 shows the whole dashboard with all the graphs laid out together.

With the data at hand, most of the above questions can be answered. Because the metrics capturing tool does not identify individual users, separate login counts for each user can not - and are not meant to - be captured. The 5-minute sum of logins shows that usually several users login within a small time period. This is expected because the application is meant to be used in groups, but can also be used individually. There are also long time periods when no users are using the service.

The timer data provides great insight to how quickly the server responds to user requests. It can be seen, that while the mean time for sending a response is only some 20ms, it has gone as high as 800ms. This means that most users get the response very fast, but few have had to wait for much longer.

This realization led to further investigation, which revealed that the logins for mobile users were fast, but the login for the web-tool was causing the slow responses. Furthermore, the response times got even higher with simultaneous login attempts from the web-tool by different users. In the end, this lead to finding and fixing a bug within the web-tool.

It can therefore be stated, that the tool for capturing business metrics functions well also in actual production. The tool was simple and straightforward to implement in the environment, and it is able to provide valuable data. The tool is versatile, allowing for several different types of data to be captured, not limiting the user to only capture pre-defined data. Also the customer has been happy with the initial results of the tool, and is eager to continue using it and extend it to cover their whole application.

6.5 Research questions

This thesis has four main research questions. With the tool implemented and tested in a production environment, they can be verified. The tests showed that the tool was well capable of capturing several different types of metrics. This was accomplished by using Counters, Meters, and Histograms. It can therefore be concluded, that by using these three general types of metrics, a wide variety of measurements can be performed. As per the first research question, these are the main types of metrics that an environment for capturing business metrics should support.

Chapters 3 and 4 of this thesis provide detailed descriptions of the requirement and the design for the tool. The tool created in this thesis is based on the ideas

created for resource monitoring systems. Both resource and business monitoring systems share similar functions. The main differences come from the type of data that they capture, the place of capture, and the time frame, as resource monitoring needs to be real-time where as business analytics often look at data from a longer historical period. This thesis shows one way of extending the resource monitoring systems to monitoring business metrics, and the author believes the method described here is functional and provides a good basis for further research. Therefore, answers to research questions 2, 3 and 4 have been found.

7 Conclusion

7.1 Summary

The aim of this thesis has been to extend current cloud resource monitoring systems to capture business metrics such as the number of logins done in the system or the average response times of the server. The motivation to do this comes from web- and mobile developer Codemate who wants to implement such a solution as part of its new development environment for cloud applications. A tool for capturing business metrics was developed to satisfy this need.

The tool for capturing business metrics in cloud applications consists of three main parts: a database, a visualization tool and a Node.js-library to gather the metrics from applications built with Node.js. The tool also includes a program written in Go-lang that provides the visualization tool with automatically created dashboards that display key metrics defined by the developer.

The database used in this thesis is InfluxDB which is a time-series database. It focuses on storing measurements which have timestamps attached to them, and to efficiently query such data. InfluxDB operates through the HTTP-protocol, making it easy to send and query data. The queries are SQL-like and can be sent as normal REST-calls together with authentication details. InfluxDB was chosen because it fits the task well and it is also used by other tools in the Codemate's new development environment.

The metrics that are stored in the database are captured by code inserted in the target application. The code is published as a separate npm- package which can be included in any Node.js project. The package is based on already existing metrics packages, but those have been modified to provide better performance and connection to the database. The package has been designed to be easy to use, and it only requires a few lines of code to initialize and operate.

The metrics are visualized with Grafana. It is a web-based visualization tool that can be used by any web-browser. Grafana provides several different visualization methods such as graphs, tables and gauges with alerts. It has several dynamic dashboards which can be customized and created by the users. Grafana was chosen for this thesis because it is one of the leading open-source visualization tools, and it is also used in other sections of Codemate's new development environment.

The database and visualization tool are run on EC2 in AWS. To provide fault-tolerance and easy replication, Kubernetes is used between the EC2 and database and visualization instances. The code library for capturing the actual measurements must be inserted directly into the monitored code.

The tool for capturing business metrics in cloud applications was developed as a software project by a single person, the author. The development started with forming the requirements for the tool through discussions with Codemate's software engineers and architects. The project proceeded to designing the system which was then implemented. The developed application was finally evaluated both against the requirements and by attaching it to an actual production server of one of Codemate's clients.

The main requirements that were identified were that the tool must take into account both the developers of the applications where the measurements are to be gathered and the users who want to see the measurements. Different types of measurements to capture and ways to visualize them were selected, and emphasis put on the performance of the system.

The design of the tool followed Kruchten's 4+1 view model [36]. It helped form the design of the different components starting from the overall view and drilling down to smaller, individual components. During design several key decisions were made such as what the database schema will be, how the metrics should be captured from the target applications and how the different components communicate with each other.

The implementation of the tool was done in two stages. In the first Stage, the tool was developed locally by using Minikube. During this Stage all the different components were created and tested that they worked together in a local environment. The second Stage consisted of moving the tool to the cloud, into AWS. During this phase the implementation from phase one was modified to work in an actual cloud environment, and the connections were again tested.

The evaluation consisted of two parts: evaluating whether the requirements were met and testing the tool in a production environment. The tool was verified to fulfill all the requirements set for it. It was then installed on the production server of one of Codemate's clients. The tool functioned just as expected, providing valuable data that has already helped improve the service. Furthermore, the client was pleased with the system and eager to continue using it.

This thesis has also managed to answer to all four research questions it set out to explore. It found good types of metrics that a monitoring environment should support and showed how current resource monitoring systems can be extended to capture business metrics. The thesis defined the requirements and design for such an environment, and finally concluded that this method provides a working and good method for capturing business metrics.

7.2 Future work

The tool created in this thesis is functional and has basic features that allow it to capture metrics in any application that is built with Node.js. There are still several improvements that can be made to the tool and how it is used. These can be divided in two categories: new features to the tool itself, and new integrations so that it is easier to use in Codemate's development environment.

The main improvements to the tool itself include new metrics libraries for different programming languages. Among JavaScript, the most important languages for Codemate are Go and Ruby. Thanks to the modular structure of the tool, creating new libraries for different languages is easy as long as they can send http-requests to add data into the database, which most of the modern languages used in web development can.

New types of measurements could also be added to the tool. It is not clear at this time what those might be, but to add them, only the metrics library needs to

be modified. New metrics can later easily be added as the need for it arises. It is expected that the functionality of the tool will grow over the years as it is used in several production applications.

The tool has two major integration tasks that it needs. Firstly, the tool is currently a separate entity that is running in an AWS stack of its own. Once the Codemate's development environment is ready, the tool created in this thesis will be attached to it. This will require some configuration of the AWS by setting up new Kubernetes clusters within the environment, as well as re-installing the tool there.

The second integration step needed is the creation of clear guidelines for identifying, selecting and implementing the metrics and their gathering from applications. Codemate has several projects running at the same time with different people working on them. All of them should however use the tool in the same manner, and know how to request new features for it. The guidelines should make it clear when to use the tool, what kinds of metrics to gather, and when to gather them.

In conclusion, it is very likely that Codemate will continue to use the tool created in this thesis. It satisfies the needs they have for capturing business metrics from cloud applications, it is easy to use and implement in the development environment and Codemate is committed to further develop the tool.

7.3 Further research

Research into gathering business metrics from cloud applications is slim at best. It is a topic that companies like Codemate and its clients are interested in. This thesis has revealed several areas where further research would be appropriate.

This thesis has implemented a method to gather business metrics by injecting the probe directly into the target application's code. While this works, in might cause problems in very large applications. Research into gathering business metrics by external probes would be helpful in such situations.

The method for implementing the metrics gathering inside the code could also be developed further. Ideally, it would not need any input from the developer at all. In order to gather metrics, a simple definition should be given, after which the system should automatically implement the gathering of those metrics on its own.

References

[1] Docker overview. https://www.docker.com/what-docker. Accessed: 28.04.2017.

[2] Grafana overview. https://grafana.com/grafana. Accessed: 03.05.2017.

[3] InfluxDB documentation. https://docs.influxdata.com/influxdb/v1.2/. Accessed: 28.04.2017.

[4] Kubernetes documentation. https://kubernetes.io/. Accessed: 28.04.2017.

[5] What are business metrics? https://www.klipfolio.com/resources/articles/what-are-business-metrics. Accessed: 18.05.2017.

[6] Iso/iec/ieee international standard - systems and software engineering – life cycle processes –requirements engineering. *ISO/IEC/IEEE 29148:2011(E)*, pages 1–94, 2011.

[7] Giuseppe Aceto, Alessio Botta, Walter de Donato, and Antonio Pescapè. Cloud monitoring: Definitions, issues and future directions. In *1st IEEE International Conference on Cloud Networking, CLOUDNET 2012, Paris, France, November 28-30, 2012*, pages 63–67, 2012.

[8] Yahya Al-Hazmi, Konrad Campowsky, and Thomas Magedanz. A monitoring system for federated clouds. In *1st IEEE International Conference on Cloud Networking, CLOUDNET 2012, Paris, France, November 28-30, 2012*, pages 68–74, 2012.

[9] Yahya Al-Hazmi and Thomas Magedanz. A flexible monitoring system for federated future Internet testbeds. In *International Conference on the Network of the Future, NOF 2012, Paris, France, November 28-30, 2012*, pages 1–6, 2012.

[10] Cloud Security Alliance. Security guidance for critical areas of focus in cloud computing v3.1. Technical report, 2017.

[11] Josef Altmann and Gustav Pomberger. Cooperative software development: Concepts, model and tools. In *TOOLS 1999: 30th International Conference on Technology of Object-Oriented Languages and Systems, Delivering Quality Software - The Way Ahead, 1-5 August 1999, Santa Barbara, CA, USA*, pages 194–207, 1999.

[12] Gabriel-Cosmin Apostol and Florin Pop. MICE: Monitoring high-level events in cloud environments. In *11th IEEE International Symposium on Applied Computational Intelligence and Informatics, SACI 2016, Timisoara, Romania, May 12-14, 2016*, pages 377–380, 2016.

[13] Xiaoying Bai, Shufang Lee, Wei-Tek Tsai, and Yinong Chen. Collaborative web services monitoring with active service broker. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference, COMPSAC 2008, 28 July - 1 August 2008, Turku, Finland*, pages 84–91, 2008.

[14] Fabio Barbon, Paolo Traverso, Marco Pistore, and Michele Trainotti. Run-time monitoring of instances and classes of web service compositions. In *2006 IEEE International Conference on Web Services (ICWS 2006), 18-22 September 2006, Chicago, Illinois, USA*, pages 63–71, 2006.

[15] Karin Becker, Duncan Dubugras A. Ruiz, Virginia S. Cunha, Taisa C. Novello, and Franco Vieira e Souza. SPDW: A software development process performance data warehousing environment. In *30th Annual IEEE / NASA Software Engineering Workshop (SEW-30 2006), 25-28 April 2006, Loyola College Graduate Center, Columbia, MD, USA*, pages 107–118, 2006.

[16] Israel Ben-Shaul and Gail E. Kaiser. A configuration process for a distributed software development environment. In *Second International Workshop on Configurable Distributed Systems, 1994, Proceedings, Pittsburgh, PA , USA, 21-23 March, 1994*, pages 123–134, 1994.

[17] Devesh Bhatt, Rakesh Jha, Todd Steeves, Rashmi Bhatt, and David Wills. SPI: An instrumentation development environment for parallel/distributed systems. In *Proceedings of IPPS '95, The 9th International Parallel Processing Symposium, April 25-28, 1995, Santa Barbara, California, USA*, pages 494–501, 1995.

[18] Jose M. Alcaraz Calero and Jaime Gutierrez. MonPaaS: An adaptive monitoring platform as a service for cloud computing infrastructures and services. *IEEE Trans. Services Computing*, 8(1):65–78, 2015.

[19] Sharon Campbell, Joe Beda, and Ken Simon. Quick start for Kubernetes by Heptio on the AWS cloud. Technical report, 2017.

[20] Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman. *Readings in information visualization - using vision to think*. Academic Press, 1999.

[21] Evellin Cristine Souza Cardoso. Challenges in performance analysis in enterprise architectures. In *17th IEEE International Enterprise Distributed Object Computing Conference Workshops, EDOC Workshops, Vancouver, BC, Canada, September 9-13, 2013*, pages 327–336, 2013.

[22] Saad Yasser Chadli, Ali Idri, Joaquín Nicolás Ros, José Luis Fernández-Alemán, Juan M. Carrillo de Gea, and Ambrosio Toval. Software project management tools in global software development: a systematic mapping study. *SpringerPlus*, 5(1):2006, 2016.

[23] Carlos Coronel and Steven Morris. *Database systems: Design, implementation, & management*. Cengage Learning, 2016.

[24] Elias Adriano Nogueira da Silva and Daniel Lucrédio. Software engineering for the cloud: A research roadmap. In *26th Brazilian Symposium on Software Engineering, SBES 2012, Natal, Brazil, September 23-28, 2012*, pages 71–80, 2012.

[25] Nelly Delgado, Ann Q. Gates, and Steve Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Trans. Software Eng.*, 30(12):859–872, 2004.

[26] Mohit Dhingra, J. Lakshmi, and S. K. Nandy. Resource usage monitoring in clouds. In *13th ACM/IEEE International Conference on Grid Computing, GRID 2012, Beijing, China, September 20-23, 2012*, pages 184–191, 2012.

[27] Neal Ford. Comparing service-based architectures. ÜberConf, 2016.

[28] Openstack Foundation. Monasca architecture. https://wiki.openstack.org/wiki/Monasca, 2015. Accessed on 2017-03-15.

[29] George Fylaktopoulos, Georgios Goumas, Michael Skolarikis, Aris Sotiropoulos, and Ilias Maglogiannis. An overview of platforms for cloud based development. *SpringerPlus*, 5(38), 2016.

[30] Luis Miguel Vaquero Gonzalez, Luis Rodero-Merino, Juan Caceres, and Maik A. Lindner. A break in the clouds: Towards a cloud definition. *Computer Communication Review*, 39(1):50–55, 2009.

[31] Ibrahim Abaker Targio Hashem, Ibrar Yaqoob, Nor Badrul Anuar, Salimah Mokhtar, Abdullah Gani, and Samee Ullah Khan. The rise of "Big Data" on cloud computing: Review and open research issues. *Inf. Syst.*, 47:98–115, 2015.

[32] Tobias Hildenbrand, Franz Rothlauf, Michael Geisser, Armin Heinzl, and Thomas Kude. Approaches to collaborative software development. In *Second International Conference on Complex, Intelligent and Software Intensive Systems (CISIS-2008), March 4th-7th, 2008, Technical University of Catalonia, Barcelona, Spain*, pages 523–528, 2008.

[33] Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Rosu. JavaMOP: Efficient parametric runtime monitoring framework. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 1427–1430, 2012.

[34] Teemu Kanstrén and Reijo Savola. Definition of core requirements and a reference architecture for a dependable, secure and adaptive distributed monitoring framework. In *The Third International Conference on Dependability, July 18-25, 2010, Venice, Italy*, pages 154–163, 2010.

[35] Seija Komi-Sirviö and Maarit Tihinen. Lessons learned by participants of distributed software development. *Knowledge and Process Management*, 12(2):108–122, 2005.

[36] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[37] Sunilkumar S. Manvi and Gopal Krishna Shyam. Resource management for infrastructure as a service (IaaS) in cloud computing: A survey. *J. Network and Computer Applications*, 41:424–440, 2014.

[38] Matthew L. Massie, Brent N. Chun, and David E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(5-6):817–840, 2004.

[39] Peter Mell and Timothy Grance. The NIST definition of cloud computing. 2011.

[40] Manoel G. Mendonça and Victor R. Basili. Validation on an approach for improving existing measurement frameworks. *IEEE Trans. Software Eng.*, 26(6):484–499, 2000.

[41] Rafael Moreno-Vozmediano, Rubén S. Montero, and Ignacio Martín Llorente. Key challenges in cloud computing: Enabling the Future Internet of services. *IEEE Internet Computing*, 17(4):18–25, 2013.

[42] Raymond J. Offen and D. Ross Jeffery. Establishing software measurement programs. *IEEE Software*, 14(2):45–53, 1997.

[43] R. S. Pressman. *Software Engineering—A Practitioner's Approach*. Palgrove Macmillan, 2005.

[44] Galen Reeves, Jie Liu, Suman Nath, and Feng Zhao. Managing massive time series streams with multiscale compressed trickles. *PVLDB*, 2(1):97–108, 2009.

[45] John. W. Rittinghouse and James. F. Ransome. *Cloud computing: Implementation, management and security*. CRC Press, Florida, 2009.

[46] Sridevi Saralaya and Rio D'Souza. A review of monitoring techniques for service based applications. In *2013 2nd International Conference on Advanced Computing, Networking and Security, Mangalore, India, December 15-17, 2013*, pages 96–101, 2013.

[47] Reijo Savola and Petri Heinonen. A visualization and modeling tool for security metrics and measurements management. In *Information Security South Africa Conference 2011, Hyatt Regency Hotel, Rosebank, Johannesburg, South Africa, August 15-17, 2011. Proceedings ISSA 2011*, 2011.

[48] Eric B. Seufert. *Freemium economics: Leveraging analytics and user segmentation to drive revenue*. Elsevier, 2013.

[49] Galit Shmuelli, Nitin R. Patel, and Peter C. Bruce. *Data mining for business analytics: Concepts, techniques, and applications with XLMiner*. John Wiley & Sons, 2016.

[50] Richard T. Snodgrass. Monitoring in a software development environment: A relational approach. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania, USA, April 23-25, 1984*, pages 124–131, 1984.

[51] Ian Sommerville. *Software Engineering (10th Edition)*. Pearson, 2015.

[52] Jonathan Spring. Monitoring cloud computing by layer, part 1. *IEEE Security & Privacy*, 9(2):66–68, 2011.

[53] Jonathan Spring. Monitoring cloud computing by layer, part 2. *IEEE Security & Privacy*, 9(3):52–55, 2011.

[54] Chien-Min Wang, Shyh-Fong Hong, Shun-Te Wang, and Hsi-Min Chen. A dual-mode exerciser for a collaborative computing environment. In *11th Asia-Pacific Software Engineering Conference (APSEC 2004), 30 November - 3 December 2004, Busan, Korea*, pages 240–248, 2004.

[55] Danping Wang. Influences of cloud computing on e-commerce business and industry. *Journal of Software Engineering and Applications*, 6:313–318, 2013.

[56] Qianxiang Wang, Jin Shao, Fang Deng, Yonggang Liu, Min Li, Jun Han, and Hong Mei. An online monitoring approach for web service requirements. *IEEE Trans. Services Computing*, 2(4):338–351, 2009.

[57] Saeed Zareian, Marios Fokaefs, Hamzeh Khazaei, Marin Litoiu, and Xi Zhang. A big data framework for cloud monitoring. In *Proceedings of the 2nd International Workshop on BIG Data Software Engineering, BIGDSE@ICSE 2016, Austin, Texas, USA, May 16, 2016*, pages 58–64, 2016.

A Database Schema: Counter

```
{
  "name": "measurement.name",
  "columns": [
      "time",
      "count",
      [tags,]
  ],
  "values": [],
}
```

B Database Schema: Meter

```
{
  "name": "measurement.name",
  "columns": [
      "time",
      "count",
      "fifteen-minute",
      "five-minute",
      "mean-rate",
      "one-minute",
      [tags,]
  ],
  "values": []
}
```

C Database Schema: Histogram

```
{
  "name": "measurement.name",
  "columns": [
      "time",
      "75-percentile",
      "95-percentile",
      "99-percentile",
      "999-percentile",
      "count",
      "max",
      "mean",
      "median",
      "min",
      "std-dev",
      "sum",
      "variance",
      [tags,]
  ],
  "values": []
}
```

D Database Schema: Timer

```
{
  "name": "measurement.name",
  "columns": [
      "time",
      "75-percentile",
      "95-percentile",
      "99-percentile",
      "999-percentile",
      "count",
      "fifteen-minute",
      "five-minute",
      "max",
      "mean",
      "mean-rate",
      "median",
      "min",
      "one-minute",
      "std-dev",
      "sum",
      "variance",
      [tags]
  ],
  "values": []
}
```

E Grafana installation

E.1 Login

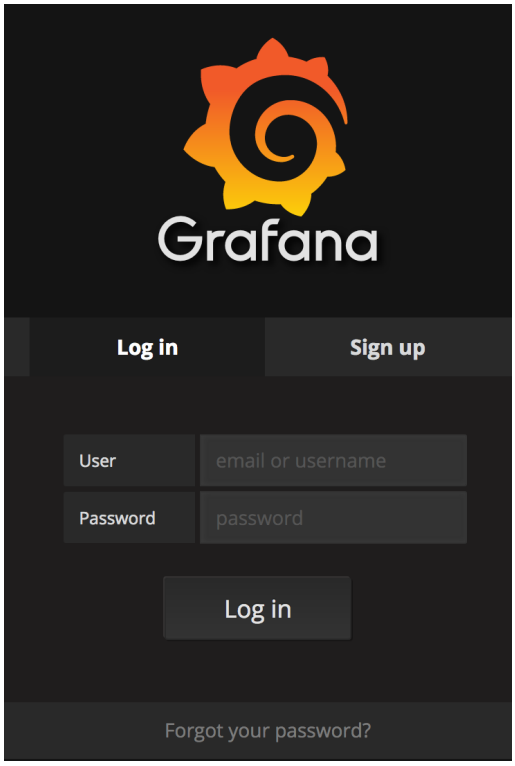

Figure E1: The login-screen for Grafana.

E.2 Add datasource

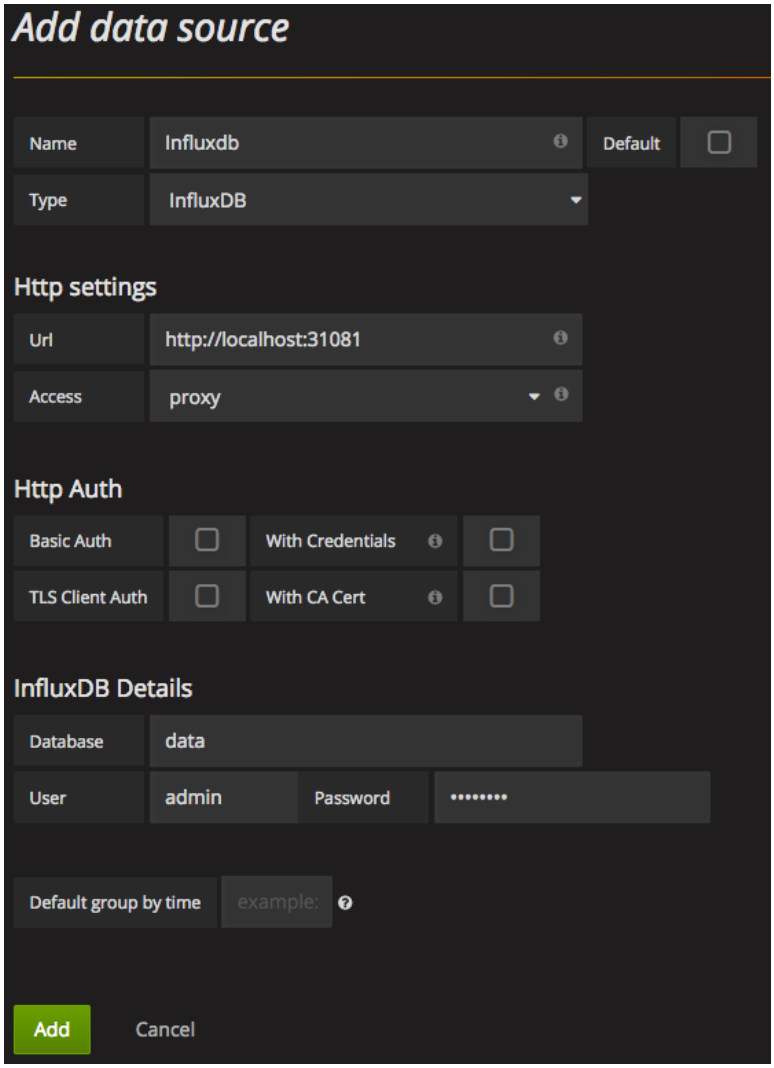

Figure E2: The datasource-screen for Grafana.

E.3 Import dashboard

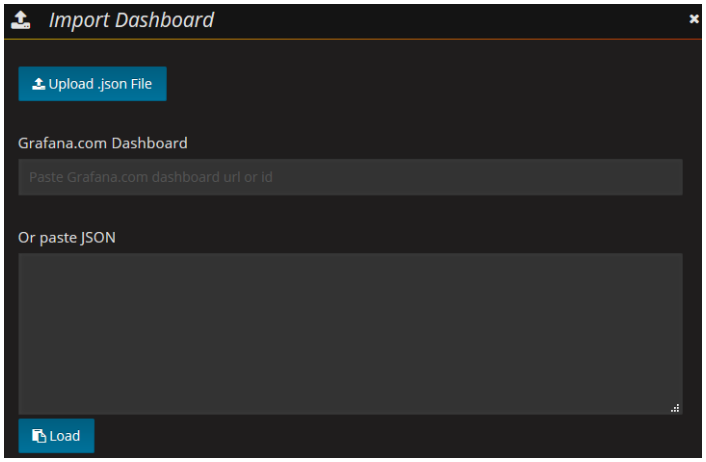

Figure E3: The pop-up screen for importing a dashboard into Grafana.

F Grafana in production

F.1 Logins today and yesterday

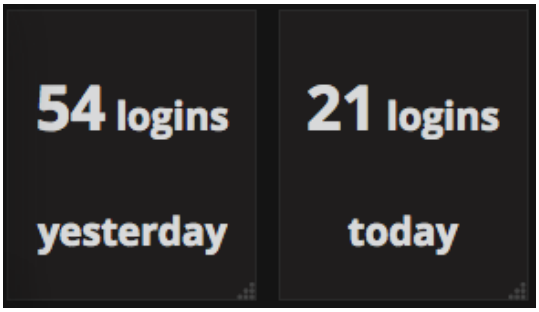

Figure F1: Graph with login amounts for today and yesterday.

F.2 Table with latest login

Login - latest events

Time	login.counter	login.timer.min	login.timer.max	login.timer.mean
2017-06-07 23:54:31	54.00	8.00	805.00	26.00
2017-06-07 23:13:44	53.00	8.00	805.00	26.00
2017-06-07 22:37:48	52.00	8.00	805.00	26.00
2017-06-07 21:29:33	51.00	8.00	805.00	26.00
2017-06-07 21:14:58	50.00	8.00	805.00	26.00
2017-06-07 20:41:34	49.00	8.00	805.00	26.00

1 2 3

Figure F2: Table with individual measurement data.

F.3 Graph with min & max & mean values

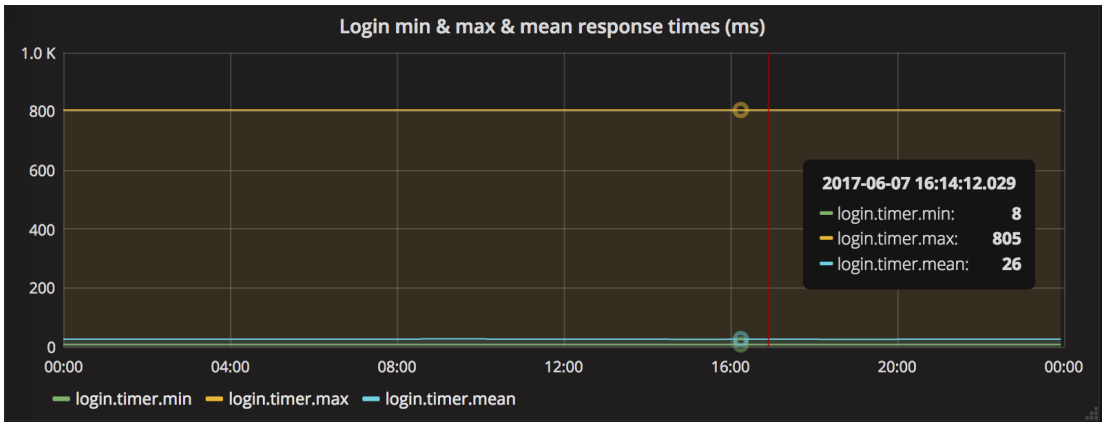

Figure F3: Graph with min & max & mean values.

F.4 Login dashboard

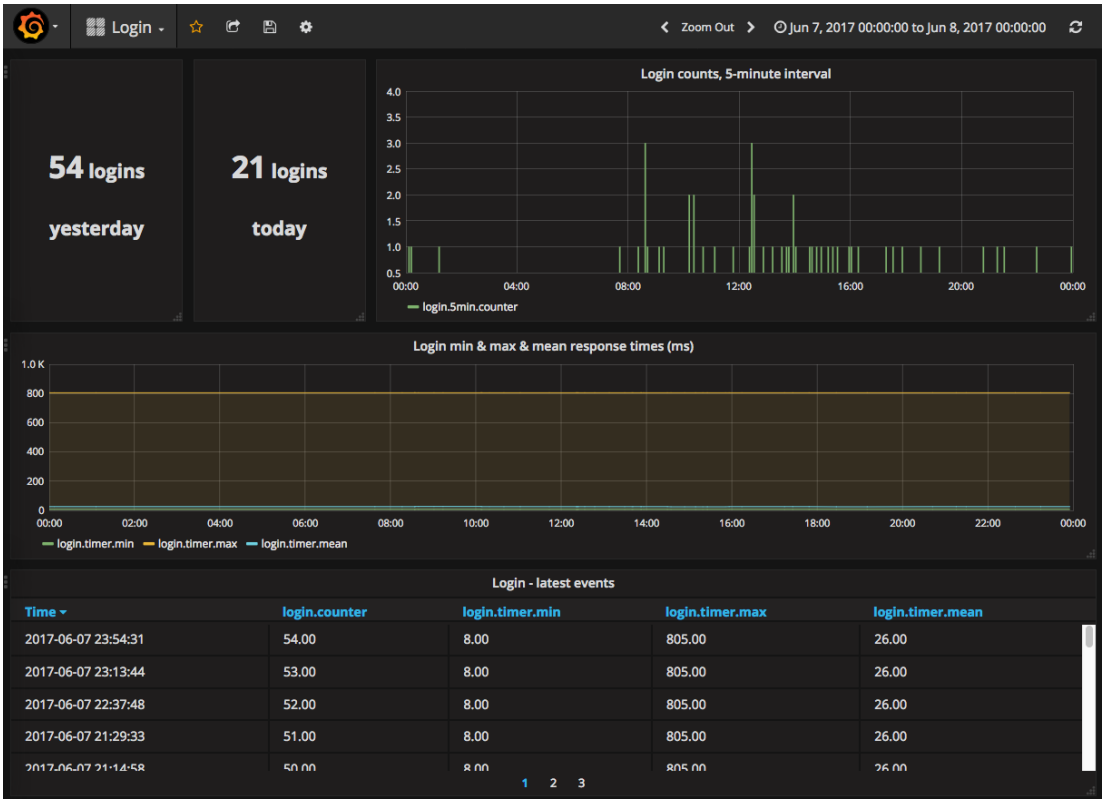

Figure F4: Table with individual measurement data.

G Database installation script on AWS

```
kubectl --kubeconfig=kubeconfig run --image=influxdb
   influxdb --port=8086 --env="INFLUXDB_HTTP_AUTH_ENABLED
   =true"
kubectl --kubeconfig=kubeconfig expose deployment
   influxdb --type=LoadBalancer --port=8086
echo Getting private IP address for the container
ipDatabase
while [ $? -ne 0 ]; do
  ipDatabase=$(kubectl --kubeconfig=kubeconfig describe
     pod influxdb | grep IP | sed -E 's/IP:[[:space:]]+//
     ')
done
echo Creating admin user
curl -s -S -i -XPOST $ipDatabase/query --data-urlencode "
   q=CREATE USER admin WITH PASSWORD 'test' WITH ALL
   PRIVILEGES" >/dev/null
echo Creating database data
curl -s -S -i -XPOST $ipDatabase/query -u admin:test --
   data-urlencode "q=CREATE DATABASE data" >/dev/null
echo Getting public IP address for the container
kubectl --kubeconfig=kubeconfig describe service influxdb
    | grep .elb.amazonaws.com
```

H Grafana installation script on AWS

```
kubectl --kubeconfig=kubeconfig run --image=grafana/
   grafana grafana --env="GF_SECURITY_ADMIN_PASSWORD=test
   "
kubectl --kubeconfig=kubeconfig expose deployment grafana
    --type=LoadBalancer --port=3000
echo Getting public IP address for the container
kubectl --kubeconfig=kubeconfig describe service grafana
   | grep .elb.amazonaws.com
```